MASTER EN SCIENCES INFORMATIQUES

Une approche génétique pour la dérivation de règles et méta-règles de transformation de modèles à partir d'exemples

Cobbaert, Quentin; Masson, Philippe

Award date:
2013

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique
Année académique 2012–2013

Une approche génétique pour la dérivation de règles et méta-règles de transformation de modèles à partir d'exemples

Quentin COBBAERT Philippe MASSON

Maître de stage : Houari SAHRAOUI

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Naji HABRA

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

La transformation de modèles est un concept central de l'ingénierie dirigée par les modèles, puisqu'elle fournit un moyen d'automatiser la manipulation de ceux-ci. Il n'est cependant pas aisé de réaliser un programme capable de transformer un modèle en un autre. Les règles qui constituent ces programmes peuvent parfois s'avérer très compliquées à trouver, même pour un expert en la matière. Une première approche pour apprendre automatiquement des règles de transformation de modèles à partir d'une paire d'exemples a été proposée par Martin Faunes. Celle-ci repose sur un algorithme génétique qui permet de dériver, de manière automatique, des programmes de plus en plus aptes à transformer des modèles. Dans le cadre de ce mémoire, nous proposons d'améliorer cette approche en introduisant une dimension de contrôle et en complexifiant la structure des règles de transformation, ce qui permettra de dériver des programmes de transformation plus performants. Deux nouvelles approches seront présentées, chacune différant dans sa manière d'apprendre le contrôle des règles. La première va générer les règles ainsi que leur ordre d'exécution simultanément tandis que la deuxième va d'abord apprendre les règles avant d'apprendre l'ordre d'exécution de celles-ci.

Mots clés: Transformation de modèles, automatique, par l'exemple, algorithme génétique

Abstract

Model transformation is a central concept of model-driven engineering since it provides a way to automate the manipulation of models. It is however not easy to create a program capable of transforming a model into another. The transformation rules are sometimes very complicated to find, even for an expert. A first approach to automatically learn model transformation rules from a set of examples by using a genetic programming-based algorithm was proposed by Martin Faunes. In the context of this thesis, we propose to improve this approach by introducing a dimension of control and by complexifying the structure of transformation rules, in order to derive more effecient transformation programs. Two new approaches are presented, each differing in the way of learning the control of the rules. The first will generate the rules and the order of execution simultaneously while the second will first learn the rules and then learn the order of execution.

Keywords: Model transformation, automated, by example, genetic algorithm

Table des matières

1 Introduction **1**
1.1 Contexte 1
1.2 Problème 3
1.3 Contribution 4
1.4 Plan 5

2 Transformation de modèles **7**
2.1 Description 7
2.2 Terminologie 8
2.2.1 Modèle 8
2.2.2 Meta-modèle 8
2.2.3 Langage de transformation de modèles 8
2.2.4 Modèle source 8
2.2.5 Modèle cible 8
2.2.6 Outil de transformation de modèles 9
2.2.7 Transformation de modèles 9
2.2.8 Transformations endogènes et exogènes 10
2.2.9 Transformations unidirectionnelles et multidirectionnelles 10
2.2.10 Transformations horizontales et verticales 11
2.2.11 Textuel et visuel 12
2.3 La transformation de modèle 12
2.3.1 Transformation Modèle-vers-Texte 12
2.3.2 Transformation Modèle-vers-Modèle 13
2.3.3 Fonctionnalités des langages de transformation 16
2.3.4 Activités au sein de la transformation de modèles . . . 18
2.3.5 Besoins pour un langage de transformation 20
2.3.6 Caractéristiques désirables d'un langage de transformation 21
2.3.7 Langages et Outils 22

2.4 Programmation par l'exemple 24
2.5 Transformation de Modèle par l'exemple 25

3 La programmation génétique 31
3.1 Description 31
3.2 Théorie 31
3.2.1 Définition 31
3.2.2 Algorithme 33
3.2.3 La fonction de fitness 33
3.2.4 Les opérateurs 34
3.3 Les phases de la programmation génétique 35
3.4 Spécification d'un problème 40
3.4.1 L'ensemble des fonction et des terminaux 41
3.4.2 La mesure de fitness 41
3.4.3 Les paramètres de contrôle 41
3.4.4 Le critère d'arrêt 42
3.5 Applications de la programmation génétique 42
3.6 Les limites de la programmation génétiques 42

4 Dérivation de règles selon un processus évolutionnaire 45
4.1 Description 45
4.2 Problème général 45
4.3 Approche initiale 50
4.3.1 Encodage des règles dans la situation initiale 51
4.3.2 Création des programmes 52
4.3.3 Opérateurs 53
4.3.4 Evaluation des programmes 54
4.4 Limites et critiques de l'approche initiale 57
4.5 Évolution de la solution selon deux pistes 63
4.5.1 Approche RAC 63
4.5.2 Approche RTC 91

5 Étude de cas 101
5.1 Description 101
5.2 Programme idéal 102
5.3 Évaluation qualitative 113
5.4 Évaluation quantitative 118
5.4.1 Évaluations multiples sur un même exemple 118
5.4.2 Évaluations sur plusieurs exemples 121
5.5 Testeur de règles 123
5.6 Discussion 124

6 Conclusion **125**
6.1 Résumé de la contribution 125
6.2 Résumé des résultats 127
6.3 Limitations 127
6.4 Travaux futurs 130

Table des figures

2.1 Types de transformations 12
2.2 Fonctionnalités des langages de transformation 16
2.3 Les différentes couches de modèles 26

3.1 Phases de la programmation génétique 35
3.2 Sélection par roulette 37
3.3 Croisement avec un seul point de croisement 39
3.4 Croisement avec plusieurs points de croisement 39
3.5 Paramètres à fournir à un algorithme génétique 40

4.1 Méta-Modèle du diagramme de classe 46
4.2 Méta-Modèle du schéma relationnel 47
4.3 Exemple de modèle de diagramme de classe 47
4.4 Exemple de modèle de schéma relationnel 48
4.5 Algorithme génétique 49
4.6 Croisement entre 2 programmes avec un point de découpage aléatoire 54
4.7 Exemple de diagramme de classe 57
4.8 Exemple équivalent à la figure 4.7 en schéma relationnel . . . 57
4.9 2ème exemple de diagramme de classe 58
4.10 Exemple équivalent en schéma relationnel 59
4.11 3ème exemple de diagramme de classe 59
4.12 Exemple équivalent en schéma relationnel 60
4.13 Exemple de schéma relationnel avec des identifiants techniques générés 61
4.14 Gestion des associations 1-1 62
4.15 Les domaines du schéma relationnel 67
4.16 Graphe des liaisons: suppression d'une condition racine . . . 75
4.17 Graphe des liaisons: deux maîtres pour un seul esclave 76
4.18 Graphe des liaisons: suppression d'une brique ayant plus d'un arc 77

4.19 Graphe des liaisons: sens de liaison possibles 77
4.20 Ensemble de terminaux non normalisé 79
4.21 Ensemble de terminaux normalisé 79
4.22 Exemple d'optimisation du NOT 82
4.23 Fusion possible entre deux classes 84
4.24 Connexions internes et externes des briques 86
4.25 Graphe d'exécution 89
4.26 Graphe d'exécution (Phase 1) 99
4.27 Graphe d'exécution (Phase 2) 99

5.1 Programme de transformation de diagrammes de classe en schémas relationnels idéal 103
5.2 Exemple de diagramme de classe 119
5.3 Exemple de schéma relationnel 119
5.4 Résultats de trois exécutions de 2000 générations de 200 programmes 120
5.5 Résultats de trois exécutions de 4000 générations de 400 programmes 120
5.6 Résultats des exemples 1 et 2 121
5.7 Résultats des exemples 3 et 4 122

Remerciements

Nous souhaitons tout d'abord remercier notre maître de stage, le Professeur Houari Sahraoui pour son excellent soutien durant les trois mois de stage que nous avons effectués à l'université de Montréal, dans les laboratoires GEODES. Travailler à ses cotés était un honneur et un véritable plaisir. Il nous a transmis sa passion pour la recherche et nous a guidés tout au long de ce parcours. Grâce à lui, nous aurons appris de nombreuses choses très intéressantes.

Nous remercions également le Professeur Naji Habra, notre promoteur, pour nous avoir offert l'opportunité d'effectuer ce stage et ce mémoire. Ce fut une expérience très enrichissante. Nous le remercions également pour les conseils qu'il nous a prodigués pour la rédaction de ce mémoire.

Merci à Martin Faunes d'avoir commencé ce travail et d'avoir bien voulu prendre de son temps pour répondre à nos questions. Ce travail était très intéressant et nous sommes fiers d'y avoir contribué.

Un grand merci aussi à l'équipe GEODES et à sa bonne humeur permanente. Ils nous ont tout de suite intégrés à l'équipe.

Nous remercions également l'ensemble du corps professoral de l'université de Namur, sans lequel nous ne serions pas arrivés à réaliser un tel travail. Nous restons convaincus que la formation universitaire de Namur est excellente.

Enfin, merci aussi à Katy Marlier pour la vérification orthographique de ce mémoire.

Chapitre 1

Introduction

1.1 Contexte

Le monde a évolué. La technologie nous entoure et nous est désormais indispensable. Et en son cœur se trouve l'informatique. Dès lors, pour que cette technologie s'améliore et évolue encore, son cœur doit battre plus vite et plus fort. Malheureusement, ce n'est pas toujours aussi simple. Dans les années 60, l'informatique, et plus particulièrement le logiciel, a connu une crise majeure [38] mettant en exergue le décalage entre l'informatique sur le terrain et ce que l'on est en droit d'attendre de celle-ci. Les systèmes informatiques attendus étaient malheureusement trop complexes à développer, à utiliser mais aussi inadéquats et non fiables. Plusieurs gros projets dans les années 70, 80 et 90 ont été victimes de cette crise et ont accusé des retards énormes (Compilateur PL1), des surcouts de production très importants (Taurus), provoqué des accidents (London Ambulance System) et ont parfois été abandonnés (Advanced Logic System). En 1995, le Standish Group Internationnal publie les résultats d'une étude portant sur le succès des développements de systèmes informatiques. Cette étude révèle que seuls 16% des projets ont pu être délivrés à temps et dans le budget prévu. 53% ont été remis hors budget et hors planning et enfin, 31% des projets ont été abandonnés en cours de réalisation.

De nos jours, la difficulté n'en est que plus importante. La puissance matérielle, le nombre d'utilisateurs, le développement d'internet sont quelques-uns des facteurs qui contribuent à rendre l'informatique de plus en plus complexe et difficile à maîtriser. Durant cette crise, l'informatique n'a cessé d'évoluer pour devenir "l'ingénierie du logiciel", passant d'une conception artisanale, à une production industrielle. De ce fait, la conception de ces systèmes informatiques requiert une grande expertise tout au long du développement, du déploiement ou de la maintenance pour assurer un

certain degré de qualité. Malheureusement, la crise perdure puisqu'encore 25% des projets actuels sont abandonnés.

L'ingénierie dirigée par les modèles

On a alors tenté de résoudre les problèmes liés à la crise du logiciel en proposant de nouvelles méthodologies, de nouveaux outils, de nouvelles procédures ou encore de nouveaux paradigmes: programmation orientée objet, architecture de composants, programmation par aspects, méthodes agiles, implication des utilisateurs dans le développement, réutilisabilité, ... Une méthodologie semble bien fonctionner: l'ingénierie du logiciel dirigée par les modèles (Model Driven Engineering, MDE). Un modèle est une abstraction de la réalité, dans le sens ou celui-ci ne peut représenter tous les aspects de cette réalité. Cela nous permet de faire face au monde d'une manière simplifiée, d'éviter sa complexité [83]. En informatique, le modèle à pour rôle de structurer les informations et activités d'une organisation. Les modèles de cycle de développement ont largement été utilisés pour réduire la complexité, les risques et pour pouvoir s'adapter au changement. Mais à l'heure actuelle, on voit de plus en plus de modèles apparaître au sein même du développement. Le langage de modélisation unifié (UML) [72], standardisé par l'Object Management Group (OMG), est couramment utilisé pour spécifier, visualiser, modifier et construire les documents nécessaires au bon développement d'un logiciel orienté objet. Il offre un standard de modélisation pour représenter l'architecture logicielle. On utilise également des modèles, comme MoCQA [97] par exemple, pour tester la qualité d'un logiciel. C'est maintenant l'ensemble du développement logiciel qui utilise des modèles. C'est pour cette raison que l'architecture dirigée par les modèles (MDA) [18], proposée et soutenue par l'OMG, tente de placer le modèle au centre du développement de systèmes informatiques, en proposant de nombreux standards regroupant notamment UML, MOF et QVT, dont nous parlerons plus tard.

Techniques au service de l'ingénierie dirigée par les modèles

Afin de développer des systèmes informatiques dans des délais et des coûts raisonnables, **l'automatisation** semble être cruciale. L'automatisation a pour but d'améliorer la qualité des logiciels en évitant certaines erreurs introduites par le facteur humain, mais permet également de gagner énormément de temps et donc d'augmenter la productivité. Par l'automatisation, on pense surtout à la génération de code source, aux tests, vérifications et validations, aux générations d'interface, etc. Différentes techniques bien connues aujourd'hui permettent d'arriver à un certain degré d'automatisation.

On peut penser notamment à **la programmation par l'exemple** qui

aura pour but de faire apprendre à un ordinateur un certain comportement selon un certain contexte. Effectivement, on peut constater que dans l'apprentissage scolaire, l'exemple a toujours été un moyen efficace pour faire comprendre des concepts, des théories, ou la façon dont un problème pouvait se résoudre. Le principe est assez simple: donner à un système informatique, des exemples montrant comment un problème peut être résolu. En lui montrant plusieurs exemples, on l'entraîne à résoudre ce problème de façon à ce qu'il puisse le faire automatiquement une fois son apprentissage terminé. Cette approche est en fait assez proche du "Machine Learning".

La programmation génétique, une sous-discipline de l'intelligence artificielle, permet de faire apprendre à une machine de nouveaux programmes grâce à des algorithmes évolutionnistes. Une fonction objectif est ainsi utilisée pour indiquer l'objectif précis que l'ordinateur doit atteindre et des opérateurs génétiques l'aideront à converger vers une solution optimale.

L'automatisation, et en particulier la programmation par l'exemple et la programmation génétique semblent donc être des techniques particulièrement intéressantes pour l'ingénierie dirigée par les modèles.

1.2 Problème

Étant donné que de nombreux modèles sont manipulés tout au long du développement de systèmes informatiques, il semblerait intéressant de pouvoir garder une cohérence très forte entre ces modèles. Cependant, les modèles changent fréquemment lors du processus de développement, pour, par exemple, répondre à de nouvelles exigences d'un client ou bien pour rectifier des erreurs de conception. Une difficulté importante est donc de maintenir cette cohérence. Les concepteurs de modèles doivent repasser sur les précédentes versions et les modifier, laissant la porte ouverte à des erreurs. Une automatisation de ce processus est donc fortement souhaitable. En automatisant les transformations modèles vers modèles et modèles vers code, on réduit l'écart entre le code source et les modèles ou entre les modèles eux-mêmes.

Bien que des avancées considérables aient eu lieu ces dernières années au niveau des outils et des environnements de modélisation, les transformations automatiques ont été restreintes à quelques applications niches telles que le "database mapping". Il n'est vraiment pas simple de transformer un modèle en un autre. L'exemple classique de transformation de modèles est celui où l'on souhaite transformer un diagramme de classe en un schéma relationnel. Cet exemple est le plus utilisé pour la simple et bonne raison que leur sémantique sont relativement proche l'une de l'autre. Une classe

correspond à une table, une association correspond à une clé étrangère. Une correspondance bidirectionnelle peut alors être clairement définie pour chaque élément. De plus, ces deux types de modèles sont largement utilisés dans le domaine et sont donc bien maîtrisés. Mais qu'en est-il si l'on doit, par exemple, transformer un modèle textuel en un modèle graphique, ou l'inverse? La correspondance est déjà nettement moins évidente. Ne serait-il pas également compliqué de transformer des types de modèles peu utilisés, dont on maîtrise parfois mal la sémantique?

La transformation de modèle est en fait bien plus compliquée qu'il n'y parait, et ce même avec des types de modèles bien connus tel que le diagramme de classe ou le schéma relationnel. La spécification des transformations de modèle est une tâche ardue pour les utilisateurs, qui doivent maîtriser les méta-modèles des modèles à transformer sur le bout des doigts, afin de couvrir toutes les possibilités et créer des transformations les plus génériques possibles, de façon à éviter des retouches à la main, parfois longues et sujettes aux erreurs. La difficulté d'écrire des règles de transformation est la motivation principale derrière cette recherche sur l'apprentissage de règles de transformation à partir d'exemple. Bien que cette idée date des années 90, le premier exemple concret de transformation de modèle par l'exemple (MTBE) a été proposé par Varrò en 2006 [99].

De plus, quand on pense à "transformation", on pense à un traitement automatique des modèles. On crée les règles de transformation, on les fournit à l'ordinateur, on appuie sur un bouton "transformer" et l'opération s'exécute pour transformer, sans intervention de l'utilisateur, un modèle source en un modèle cible. Mais souvent, il est nécessaire de performer des tâches avant d'autres pour garantir un certain résultat. L'ordinateur a donc besoin qu'on lui spécifie un ordre dans lequel il pourra exécuter ses transformations.

1.3 Contribution

Le but de ce mémoire est de poursuivre le développement d'une approche de transformation de modèle par l'exemple. Cette approche soulage l'utilisateur de la conception des règles de transformation grâce à l'exploitation d'exemples, qui permettent à l'ordinateur de dériver lui-même des règles de transformation. L'utilisateur n'a alors plus qu'à imaginer des exemples de transformations qui permettront à l'ordinateur de dériver des règles de transformation qui satisferont l'exemple imaginé.

La programmation génétique est au cœur de ce travail. C'est grâce à

cette technique qu'il est possible de dériver automatiquement des règles de transformation et de faire converger le programme vers une solution idéale, grâce à divers opérateurs génétiques qui améliorent les programmes de transformation de modèles de génération en génération, comme l'évolution a amélioré les espèces pour les rendre plus aptes à leur environnement.

L'approche initiale, développée par Martin Faunes, permet de dériver automatiquement des plans de transformation de modèles, sur base d'une paire d'exemples fournie en entrée avant chaque exécution [33]. Ce programme prend en charge des transformations de diagrammes de classe vers des schémas relationnels, mais aussi des transformations de diagrammes de séquence (complexes) vers des machines à états. Au travers de ce mémoire, une analyse de cette approche existante sera effectuée et permettra d'identifier ses limites et des améliorations seront proposées afin de les effacer ou de les atténuer. Parmi ces limitations, nous verrons que les règles générées sont un peu trop simplistes et qu'elles ne permettent pas d'arriver à des plans de transformation suffisamment complexes pour pouvoir traiter des modèles relativement complets et proches de la réalité de manière satisfaisante. En effet, les méta-modèles utilisés dans l'approche existante sont fortement simplifiés pour les besoins du travail. La structure des règles de transformation sera donc revue et permettra de générer des règles de transformation plus complexes, composées de plusieurs conditions et d'autres propriétés.

L'un des problèmes clé de ce travail consistera à mettre au point un mécanisme permettant de contrôler le flux d'exécution des règles, nécessaire pour prendre en charge des transformations plus complexes. Plusieurs mécanismes de contrôle seront étudiés et testés, afin de permettre à l'application de converger le plus efficacement vers la solution optimale.

La solution initiale évoluera selon deux axes, différant dans leur façon de trouver un ordonnancement optimal des règles. Une des deux approches apprendra les règles de transformations et le contrôle de manière simultanée, tandis que la deuxième approche apprendra dans un premier temps les règles de transformations, et dans un second temps l'ordonnancement des règles.

1.4 Plan

Le document est structuré de la manière suivante:

Le chapitre 2 présente la transformation de modèles, en clarifiant tout d'abord quelques notions tournant autour de ce sujet. Ensuite, un état de l'art présentera les différentes études en cours dans ce domaine, mais aussi les langages et outils qui ont été développés au cours des dernières années.

Un état de l'art de la programmation par l'exemple et d'autres techniques "par l'exemple" seront également présentées. Grâce à cela, nous pourrons voir comment se positionne ce travail.

La programmation génétique constitue le cœur de ce travail. Dès lors, le chapitre 3 présente la programmation génétique en général et les diverses applications et travaux qui ont pu être réalisés grâce à cette technique. Son mode de fonctionnement sera expliqué en détail et ses limites seront exposées en fin de chapitre.

Le chapitre 4 constitue la plus grosse partie de ce mémoire. L'approche initiale, sur laquelle se base ce mémoire, ainsi que ses limites et faiblesses seront analysées. A partir de là, de nombreuses améliorations seront proposées. Une autre approche sera également présentée. Celle-ci diffère de la précédente dans la façon de chercher un ordonnancement optimal des règles.

Le chapitre 5 propose une étude de cas, dans laquelle sont proposées une évaluation qualitative et quantitative de la nouvelle version de l'approche, afin de comparer ses résultats avec la précédente.

Le chapitre 6 conclut ce travail en le positionnant par rapport aux approches et outils présentés dans le deuxième chapitre. Ses limites seront identifiées et de futures améliorations qui permettront d'améliorer les résultats et la qualité générale des plans de transformation générés seront proposées.

Chapitre 2

Transformation de modèles

2.1 Description

Ce chapitre a pour objectif de positionner ce mémoire par rapport aux approches, langages et outils déjà développés dans le cadre de la transformation de modèles. Dans ce chapitre, une brève introduction à la transformation de modèle sera proposée. Les concepts majeurs, tels que les modèles, les méta-modèles, ou encore les différents types de transformation seront définis clairement. La transformation de modèles est divisée en plusieurs catégories. Chacune d'entre elles sera définie et plusieurs approches et travaux relatifs à ces catégories seront présentés. Nous verrons ensuite quelles sont les fonctionnalités qui constituent les langages de transformation de modèles grâce à un feature diagram. Cette partie sera complétée par une autre analyse qui propose une liste de caractéristiques désirables pour les langages de transformation. Nous verrons également les activités au sein de la transformation de modèles et les besoins qu'un langage de transformation doit respecter pour être utile et pratique. Pour conclure cette présentation sur la transformation de modèles, une présentation des langages et outils phares sera proposée. Via cette celle-ci, nous verrons certains de leurs avantages et certaines de leurs faiblesses. Pour terminer ce chapitre, nous présenterons également la programmation par l'exemple et plus spécifiquement la transformation de modèles par l'exemple (MTBE). Cette partie aidera le lecteur à comprendre l'intérêt de l'utilisation d'exemples. Divers travaux utilisant des exemples seront présentés en fin de chapitre.

2.2 Terminologie

Avant de développer la théorie liée à la transformation de modèles, il est nécessaire de clarifier les différentes notions qui apparaîtront dans celle-ci.

2.2.1 Modèle

En informatique, un modèle a pour objectif de structurer les informations et activités d'une organisation: données, traitements, et flux d'informations entre entités. Les modèles actuellement employés dans les entreprises et les administrations restent souvent inspirés des modèles développés dans les années 1970 (modèle entité-relation). Un modèle est une représentation simplifiée d'un système qui permet de rendre la compréhension de celui-ci plus aisée [85]. La plupart des modèles sont graphiques mais peuvent également être textuels et sont exprimés dans des langages spécifiques à des domaines ou dans des langages de modélisation tels qu'UML.

2.2.2 Meta-modèle

Un méta-modèle est un modèle de modèle. Il définit un ensemble de concepts d'un certain domaine, et les relations qui pourraient apparaître entre eux. Il décrit la structure de modèles que ceux-ci doivent respecter afin d'être valides. Les méta-modèles servent également de base pour construire des éditeurs de modèles, des analyseurs de modèles ou des générateurs de code pour des domaines spécifiques [37]. Un méta-modèle peut être vu comme une grammaire pour modèles.

2.2.3 Langage de transformation de modèles

Un langage de transformation de modèles est un vocabulaire et une grammaire qui permettent de créer des transformations de modèles. Grâce à ce langage, des règles de transformation pourront être écrites par l'utilisateur ou dérivées par une machine.

2.2.4 Modèle source

S'il y a une transformation, cela veut dire qu'un modèle doit être fourni en "entrée" à cette transformation. Le modèle fourni en entrée est appelé communénement "modèle source". Il doit être conforme à un méta-modèle source.

2.2.5 Modèle cible

De même, si on transforme un modèle, cela veut dire qu'on va obtenir un autre modèle, résultat de la transformation. Il s'agit de la sortie de la trans-

formation et on l'appelle généralement le "modèle cible". Bien évidemment, il doit être conforme à un méta-modèle cible.

2.2.6 Outil de transformation de modèles

Un outil de transformation de modèles exécute une transformation sur un modèle source afin de produire un modèle cible. GReAT [15] ou ATL [46] sont tous deux des outils de transformation de modèles dont nous parlerons plus loin.

2.2.7 Transformation de modèles

La transformation de modèles, utilisée dans l'ingénierie dirigée par les modèles, vise a réduire l'effort et les erreurs en automatisant la construction ou la modification de modèles quand cela est possible.

Kleppe et al. [100] définissent une transformation de modèle comme étant une génération automatique d'un modèle cible à partir d'un modèle source. La définition d'une transformation est un ensemble des règles de transformation qui définissent, ensemble, comment un modèle défini dans un langage source peut être transformé dans un modèle défini dans un langage cible. Une règle de transformation est une description de la façon dont une ou plusieurs constructions du langage source peu(ven)t être transformée(s) en une ou plusieurs constructions du langage cible.

En fait, la transformation de modèle peut être vue comme une sorte de programme qui prend un modèle en entrée, et fournit un autre modèle en sortie, ou encore comme un programme qui fait muter un modèle en un autre [95]. Cependant, il existe de nombreuses variétés de transformations de modèles, différant dans leur utilisation ou encore dans les paramètres d'entrée et sortie qu'elles utilisent. Mais de façon générale, une transformation de modèle spécifie habituellement quels modèles sont acceptables en entrée et quels modèles elle peut produire comme sortie lorsque cela est approprié.

Selon Mens et al. [67], le nombre de modèles fournis en entrée ou en sortie peut varier. Nous conviendrons toutefois qu'il est nécessaire de fournir au moins un modèle en entrée. Il se peut qu'une transformation de modèles ne produise pas de modèle de sortie, mais on parlera alors d'analyse de modèle.

Dans le cas où l'on utilise plusieurs modèles en entrée, on peut parler de fusion de modèles, où il est question, par exemple, de fusionner plusieurs modèles sources développés en parallèle en un modèle résultant. Enfin, on peut imaginer une transformation qui prendrait un modèle indépendant

d'une certaine plateforme (Platform Independent Model, PIM) en entrée, et le transformerait en une série de modèles spécifiques à une certaine plateforme (Platform Specific Model, PSM).

2.2.8 Transformations endogènes et exogènes

En fonction du méta-modèle dans lequel les modèles cible et source sont exprimés, une transformation peut être de type endogène ou de type exogène. Une transformation de modèles exogène est une transformation entre modèles ayant des méta-modèles distincts. Un exemple simple est celui où l'on souhaite transformer un diagramme de classe UML en un schéma relationnel. Comme exemple de transformations exogènes on retrouve notamment:

- La synthèse d'un modèle de haut niveau, plus abstrait, vers un modèle de plus bas niveau, plus concret. Un exemple de synthèse est la génération de code. Par exemple, on peut transformer un schéma relationnel en code DDL.
- Le reverse engineering, qui est en fait l'inverse de la synthèse. Il s'agit ici de remonter vers un niveau supérieur, plus abstrait, à partir d'un niveau inférieur.
- La migration d'un programme écrit dans un langage vers un autre langage tout en conservant le même niveau d'abstraction.

Une transformation endogène, quant à elle, transforme des modèles issus d'un même méta-modèle. Comme exemple de transformations endogènes, on peut citer:

- L'optimisation, où l'on essaye d'améliorer certaines qualités opérationnelles d'un programme en préservant sa sémantique.
- Le refactoring, une technique consistant en la restructuration d'un morceau de code, alternant ainsi sa structure interne mais ne changeant pas son comportement externe.
- La normalisation, utilisée pour atténuer la complexité syntaxique, en transformant du sucre syntaxique en des constructions plus simples du langage.

2.2.9 Transformations unidirectionnelles et multidirectionnelles

Il est également important de distinguer la notion d'unidirectionnalité et de bidirectionnalité des transformations de modèles. Une transformation de modèles unidirectionnelle n'a qu'un seul mode d'exécution: elle prend toujours le même type de modèle en entrée et produit le même type en

sortie. Ce type de transformation sera, par exemple, utile dans le cas de la compilation. Dans le cas d'une transformation de modèles bidirectionnelle, un même type de modèles peut, tantôt être utilisé en entrée, tantôt être utilisé en sortie. Ce genre de transformation sera utile dans le cas d'un travail sur plusieurs modèles qui doivent rester consistants entre eux. Par exemple, le changement dans un modèle implique le changement dans un autre modèle afin de maintenir cette consistance.

2.2.10 Transformations horizontales et verticales

Une transformation horizontale est une transformation où les modèles cibles et sources se trouvent au même niveau d'abstraction. Le refactoring est un exemple de transformation horizontale. Une transformation verticale est une transformation où les modèles cibles et sources se trouvent à des niveaux d'abstraction différents. Le raffinement, lorsque l'on implémente des spécifications par exemple, en est un exemple.

Une transformation est toujours verticale ou horizontale et toujours endogène ou exogène, comme le montre le schéma suivant (fig. 2.1). Les transformations portent ainsi un nom plus précis lorsqu'on identifie leurs propriétés.

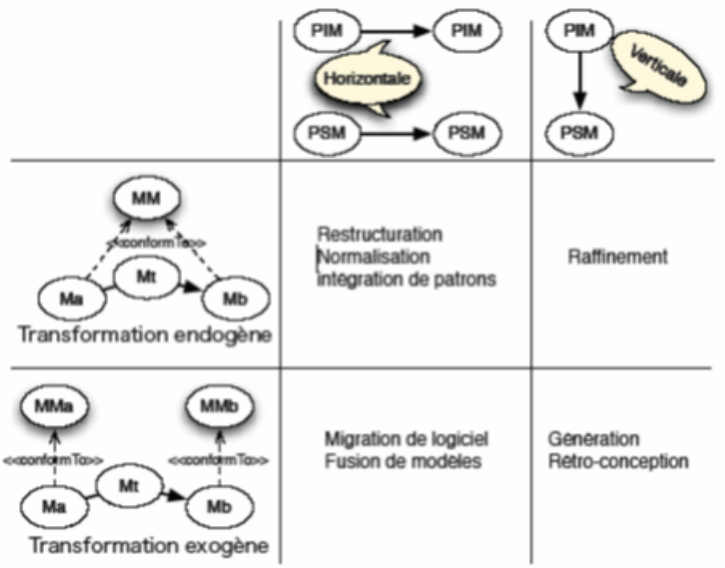

Figure 2.1: *Types de transformations*

2.2.11 Textuel et visuel

Certaines approches de transformation de modèles sont tantôt visuelles, tantôt textuelles. ATL ou Kermeta [45] sont des exemples d'approches textuelles puisqu'il est nécessaire de spécifier les transformations par écrit, au contraire des approches visuelles qui les spécifient de manière graphique. QVT [69] propose les deux manières de faire dans son langage QVT Relational.

2.3 La transformation de modèle

La transformation de modèles peut être divisée en deux sous-catégories selon Czarnecki et Helsen [28] : les transformations Modèle-vers-Code et les transformations Modèle-vers-Modèle. Cependant, étant donné que du code source est, la plupart du temps, du texte, nous parlerons plus généralement d'approche Modèle-vers-Texte.

2.3.1 Transformation Modèle-vers-Texte

Les approches Modèle-vers-Texte (Model-to-Text, M2T) ont pour but de transformer des modèles en texte. Il s'agit par exemple de produire de la documentation à partir d'un modèle, comme un diagramme de classe UML par exemple. Deux types d'approche existent au sein de la transformation de modèles en texte:

- Approches "Visitor-based": Grâce à des mécanismes d'exploration de la représentation interne des modèles, il est possible d'écrire du "texte", la plupart du temps du code source comme le permet Jamda [10]. Cependant, Jamda ne supporte pas MOF [73], le standard défini par OMG, permettant de modifier les méta-modèles. D'autres outils et approches existent.

- Approches basées sur des patrons: Les outils tels que openArchitectureWare [19], JET [63], Codagen Architect [1], AndroMDA [8], ArcStyler [9], MetaEdit+ [3], et OptimalJ [4] utilisent des patrons pour générer du code. Un patron consiste généralement en du texte contenant des bouts de meta-code qui permettent d'accéder aux informations de la source et d'opérer une sélection de code dans celui-ci. Ce sont des règles qui contiennent, d'une part, un coté gauche, composés de logique exécutable pour accéder au code, et , d'autre part, un coté droit qui combine des chaines de caractères à produire avec de la logique pour la sélection de code.

2.3.2 Transformation Modèle-vers-Modèle

Les approches Modèle-vers-Modèle (Model-to-Model, M2M) sont celles qui sont les plus proches de ce mémoire. Les approches M2M transforment des modèles sources en modèle cibles. Les transformations peuvent traiter des modèles de méta-modèles différents ou de mêmes méta-modèles. Malgré le fait que le M2T soit le plus utilisé dans le MDA, et en particulier le Model-to-Code, le M2M est nécessaire pour beaucoup d'aspects de l'ingénierie du logiciel, tel que la documentation. La documentation est trop souvent laissée à l'abandon mais est pourtant cruciale dans le développement du logiciel. Avec le M2M, les modèles d'un même système informatique (exigences, composants, code source, etc) restent cohérents entre eux, et sont donc plus facilement modifiables, maintenables, et peuvent plus facilement être optimisés.

Approches par manipulation directe

Les approches par manipulation directe fournissent des représentations de modèles et des APIs pour les manipuler. C'est donc aux utilisateurs d'implémenter les règles de transformation, le traçage, ou encore le contrôle à partir de zéro. Ces APIs sont pour la plupart implémentées dans des langages orientés objets. Jamda fait partie de cette sous-catégorie. Builder Object Network (BON) est un outil de manipulation directe. Il fournit un réseau d'objet C++ grâce auxquels il est possible de mettre à jour et de naviguer dans des modèles.

Approches déclaratives

Les approches déclaratives regroupent les approches utilisant des relations mathématiques et les approches déclaratives. Des relations entre les éléments des modèles sources et cibles sont spécifiées avec des contraintes (prédicats). On peut alors voir les approches relationnelles comme un problème d'optimisation. Gerber et al. ont utilisé Mercury [87], un langage de programmation logique, pour implémenter des transformations. QVT par QVT-Partners (QVTP) [80] est un exemple d'approche relationnelle. Ce genre d'approche est généralement plus facile à comprendre et à écrire et permet, entre autre, de naviguer dans le modèle source, la traçabilité des transformation et la bidirectionnalité des transformations. De plus, les approches relationnelles n'introduisent pas d'effet de bord.

Approches impératives

Les approches impératives sont des approches qui permettent de contrôler le séquencement d'une transformation. Les approches impératives se concentrent sur comment une transformation doit être effectuée en spécifiant les étapes requises pour dériver le modèle cible à partir du modèle source. Ce genre d'approche est utilisée pour les transformations où l'approche déclarative échoue, et plus spécifiquement quand un contrôle explicite est nécessaire dans l'ensemble des transformations. L'approche impérative fournit des moyens de séquencement des transformations, de sélection et d'itération.

Approches basées sur les transformations de graphes

Les approches basées sur les transformations de graphes reposent sur la théorie des graphes. Les modèles à transformer sont représentés sous forme d'arbre. L'arbre syntaxique abstrait du modèle source est transformé suivant des règles et l'arbre syntaxique abstrait résultant est celui du modèle cible. Les règles sont constituées de deux parties (gauche et droite) et chaque partie est constituée de patrons de graphes. La partie gauche de la règle doit matcher avec le modèle devant être transformé et est alors remplacé par la partie droite de la règle. Des conditions peuvent accompagner le patron dans la partie gauche, ainsi que des opérateurs logiques. GReAT, VIATRA [27], AToM3 [31], UMLX [103] et BOTL [20] sont des approches utilisant les transformations de graphes.

Les Triple Graph Grammars (TGG) [52], sont des grammaires qui permettent de décrire des transformations de graphes. Leurs règles sont spécifiées au moyen de trois graphes:

- Graphe de gauche: sous-graphe du graphe source. Il s'agit des préconditions de la règle.

- Graphe de droite: sous-graphe du graphe cible. Il s'agit des post-conditions de la règle.
- Graphe de correspondance: décrit les mappings entre les éléments du graphe de gauche et du graphe de droite.

Approches dirigées par la structure

Les approches dirigées par la structure fonctionnent en deux phases:

1. Créer la structure du modèle cible
2. Assigner les attributs et les références dans le modèle cible.

OptimalJ implémente cette approche en Java. Le framework s'occupe de la planification et de l'application des règles, mais ce sont les utilisateurs qui doivent fournir celles-ci. QVT par Interactive Objects and Project Technology (IOPT) [94] est une approche dirigée par la structure. Une règle est définie par des éléments du modèle cible et l'emboitement des règles correspond à la hiérarchie des éléments du méta-modèle cible. On peut voir cette approche comme une configuration top-down du modèle cible.

Approche hybrides

Les approches hybrides sont des approches qui fournissent un langage déclaratif et impératif pour décrire les règles de transformation. C'est à l'utilisateur de choisir la méthode qu'il préfère. ATL et XDE [5] sont des exemples d'approches hybrides. QVT fournit deux langages différents: QVT relational (déclaratif) et QVT Operational (impératif).

2.3.3 Fonctionnalités des langages de transformation

Czarnecki et Helsen fournissent un feature diagram représentant les différentes composantes de la transformation de modèle (fig. 2.2).

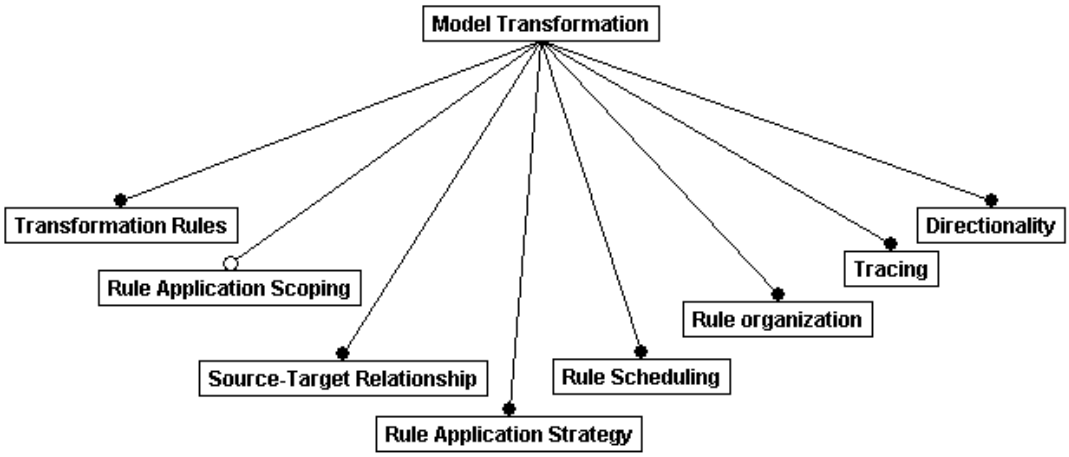

Figure 2.2: *Fonctionnalités des langages de transformation*

Cependant, la plupart des approches existantes ne couvrent pas l'entièreté de ces caractéristiques et se concentrent sur quelques-unes d'entre-elles. Celles qui nous intéressent dans le cadre de ce mémoire sont les règles de transformation, les relations entre les modèles sources et cibles, les stratégies d'application des règles, la planification des règles, l'organisation des règles et la directionnalité.

Dans la plupart des approches existantes, les règles sont constituées de deux parties: la partie gauche (Left-Hand Side, LHS) et la partie droite (Right-Hand Side, RHS). Le LHS est dédié au modèle source, tandis que le RHS est dédié au modèle cible. Ces deux parties contiennent des variables, des patrons (des chaines de caractères ou des patrons graphiques), des opérateurs logiques, tels que le OR, le AND, le NOT, mais également des requêtes telles que des OCL-queries [74] permettant de retrouver des éléments du modèle source.

Le périmètre d'application des règles (Rule application Scoping) permet de réduire les parties d'un modèle qui participe à une transformation. Certaines approches permettent de configurer un périmètre de modèle source (XDE et GReAT) et de modèle cible (XDE).

Même si par transformation de modèles on entend modèles de méta-modèles distincts, certaines approches ne permettent de faire que du refactoring, c'est-à-dire d'améliorer un modèle. Mais dans ce cas, le modèle cible de la transformation partage le même méta-modèle que le modèle source. C'est le cas de VIATRA et GReAT. Certaines approches telles que XDE permettent le fait que le modèle cible soit un nouveau modèle.

Un autre aspect important est la stratégie d'application des règles. Dans certains cas, il peut y avoir plusieurs éléments pouvant activer une règle. Il faut alors pouvoir définir une stratégie d'application de ces règles. Cette stratégie est soit déterministe, c'est-à-dire qu'elle s'appliquera toujours d'une certaines façon; soit non-déterministe, c'est-à-dire que la règle peut s'appliquer à un des cas qui l'a déclenché, au hasard.

La planification des règles permet de déterminer un ordre d'application parmi les règles. Cet ordre peut être déterminé de façon implicite ou de façon explicite. Dans le cas d'un ordre implicite, l'utilisateur n'aura aucun contrôle sur les règles. Le moteur d'inférence va choisir lui-même l'ordre d'application des règles. Pour cela il utilise un agenda. C'est le cas d'OptimalJ. Il est cependant possible que l'utilisateur "dirige" l'exécution en utilisant des éléments du langage pour le forcer à exécuter des règles dans un certain ordre. On peut imaginer que la première règle appelle la deuxième dans son côté droit, et que la deuxième appelle la troisième et ainsi de suite. Si par contre le contrôle est explicite, l'utilisateur dispose de moyens pour définir un ordre d'exécution. En cas de conflit, les outils proposent parfois des mécanismes de résolution de ces conflits. Certaines approches proposent de laisser l'utilisateur choisir (XDE).

L'organisation des règles s'occupe du rassemblement de règles ou de la structuration de groupes de règles. Certains outils, comme JESS [42] ou VIATRA, proposent un mécanisme permettant de rassembler les règles en modules. D'autres permettent des mécanismes d'héritage entre modules ou entre règles comme QVTP. Aussi, les règles peuvent être organisées selon la structure du modèle source ou du modèle cible comme QVT par IOPT, où une règle est définie pour chaque élément du modèle cible. Ces règles sont également emboitées en fonction de la hiérarchie des éléments définis dans le méta-modèle cible.

Les transformations peuvent enregistrer les liens entre les éléments sources et les éléments cibles. Ces liens peuvent être utiles pour effectuer une analyse d'impact (analyser comment la modification d'un modèle pourrait en affecter d'autres) ou pour la synchronisation entre modèles. Plusieurs approches fournissent des mécanismes de traçabilité comme QVT par CDI [24] et QVT par IOPT, tandis que d'autres comptent sur l'utilisateur pour

"encoder" les traces eux-mêmes, comme VIATRA et GReAT. Certaines approches requièrent un encodage manuel des liens de traçabilité dans les règles de transformation par les développeurs, comme QVT par CDI. Dans le cas d'un support automatisé comme avec QVT par IOPT, l'approche peut fournir plusieurs contrôles sur la façon dont les liens peuvent être créés, de façon à limiter la quantité de données de traçabilité. Enfin, il faudra choisir où stocker les liens de traçabilité: dans le modèle source, dans le modèle cible, ou séparément.

Enfin, les transformations peuvent être unidirectionnelles ou bidirectionnelles. Les transformations unidirectionnelles ne peuvent être exécutées que dans une seule direction. Dans ce cas, un modèle cible est calculé à partir d'un modèle source. Les transformations bidirectionnelles peuvent, elles, être exécutées dans les deux sens, ce qui est très pratique pour la synchronisation de modèles par exemple. Bien entendu, la bidirectionnalité peut être accomplie de deux manières: en utilisant une règle bidirectionnelle ou en utilisant deux règles unidirectionnelles complémentaires, une pour chaque direction. Peu d'approches fournissent des moyens de créer des règles bidirectionnelles, généralement plus difficiles à implémenter. L'approche d'Akehurst et Kent [12] et QVTP le permettent néanmoins.

2.3.4 Activités au sein de la transformation de modèles

Selon Mens [66], la transformation de modèles couvre de nombreuses activités. Cette section en décrit quelques-unes et présentent des travaux y étant associés.

Génération automatique de code

L'une des activités les plus recherchées dans le milieu de l'ingénierie dirigée par les modèles est la génération de code. Celle-ci permet un gain de productivité énorme et évite des erreurs de programmation parfois coûteuses en temps et en argent. Le but est la transformation de modèles indépendants de plateformes particulières (PIM) en des modèles destinés à une plateforme spécifique (PSM) et, au final, de générer du code source à partir de modèles. Executable UML [65] est une approche prometteuse à ce niveau puisqu'elle permet une génération complète de code à partir de modèles UML. On peut également désirer faire l'opération inverse, à savoir extraire un modèle à partir de code source. Cette technique est appelée l'extraction de modèle est très utile pour le reverse engineering [70].

Traduction de modèles

La traduction de modèle est le type d'approche le plus fréquemment abordé dans littérature. Le but est de transformer un modèle exprimé dans un

certain langage dans un autre langage de modélisation. Class2RDBMS [41] en est un exemple. Il permet de transformer un diagramme de classes vers un schéma relationnel. Un autre but est également de gérer les problèmes sémantiques d'un langage de modélisation en exprimant les modèles dans d'autres langages plus formels [14]. Straeten et al. [91] ont essayé de traduire des modèles UML en logique descriptive. De plus, il est parfois souhaitable de pouvoir manipuler un même modèle avec plusieurs outils de modélisation différents. La transformation de modèle UML vers XMI, le standard OMG basé sur XML pour les conversions de modèles [75], permet notamment l'échange de modèles UML entre différents outils de modélisation UML.

Exécution de modèles

Aussi, si certains modèles servent à représenter un comportement, il peut être intéressant d'exécuter ce comportement afin de valider le modèle. C'est ce qu'étudient les approches de simulation ou d'exécution de modèles. Certaines de ces approches transforment des modèles en des systèmes exécutables via des compilateurs/interpréteurs comme Executable UML [81]. Cependant, ces approches requièrent parfois l'intervention de l'utilisateur pour fournir des informations que le simulateur n'aurait pas pu déduire. Störrle et Hausmann [90] permettent de simuler le comportement de diagrammes d'activités UML en les transformant dans un premier temps en réseaux de Petri [77] puis en l'exécutant dans un deuxième temps.

Vérification et validation de modèles

La vérification de modèles est également sujette à diverses recherches. Il est en effet désirable de pouvoir vérifier et valider des modèles, afin de voir s'ils sont conformes aux exigences ou s'ils sont corrects syntaxiquement et sémantiquement. Plusieurs langages de vérification de modèles existent: Umltocsp [23] ou encore Automated formal verification of visual modeling languages by model checking [98]. Il est également intéressant de pouvoir vérifier et valider les transformations de modèles en elles-mêmes, c'est-à-dire établir si une transformation à un sens et si elle est utile. Narayanan et Karsai [71] et Baudry et al. [16] proposent le test et la vérification de transformations de modèles. Ce genre de vérification est notamment utilisé pour s'assurer que des transformations préservent la correction et la consistance de modèles. Stell et Lawley [88] expliquent comment valider le moteur de transformation de modèles Tefkat [61]. Varró [98] et Küster [59] se chargent d'étudier la vérification formelle de transformations de modèles exprimées à l'aide de règles de transformation de graphes. Anastasakis et al. [14] utilisent le langage et outil (solveur de contraintes) Alloy [44] pour vérifier des transformations de modèles.

Évolution et maintenance des modèles

L'évolution des modèles ou des langages de modélisation fait également l'objet de recherches. Lorsqu'un langage de modélisation change, et plus spécifiquement son méta-modèle, les utilisateurs sont contraints de migrer leurs modèles vers la nouvelle version du langage et donc de les transformer mais aussi de transformer les transformations en elles-mêmes. C'est le domaine de la co-évolution des modèles. Ce domaine a été étudié par Wachsmuth [102].

ATL est parvenu à gérer ce problème en utilisant un transformateur de transformation automatique, fournissant ainsi une maintenance automatique des transformations en cas de modification des méta-modèles utilisés.

Étant donné que des modèles peuvent être encodés au format XML grâce a XMI, il est dès lors possible d'utiliser XSLT [6] pour transformer des fichiers XML. Cependant, une telle approche est limitée à cause de son manque de mise à l'échelle. Le fait de modéliser des transformations à la main en XSLT les rend rapidement difficilement maintenables à cause du caractère très verbeux de XML et XSLT. C'est pour cette raison que Peltier et al. [76] ont proposé une approche permettant de générer des règles de transformation XSLT. Cependant cette approche n'a pas donné les résultats escomptés.

Amélioration de la qualité des modèles

La qualité d'un modèle est également importante, et celle-ci se dégrade au cours du temps étant donné que les modèles évoluent durant leur vie. Le model refactoring permet d'améliorer la qualité d'un modèle, en modifiant par exemple sa structure de façon à le rendre plus lisible. Il s'agit donc d'une transformation endogène qui modifie la structure tout en préservant son comportement. Divers auteurs ont commencé à explorer le problème de refactoring de modèles UML. ([79], [26], [64], [96], [105])

2.3.5 Besoins pour un langage de transformation

Selon Pollet [78], pour être utile et pratique, un langage de transformation doit répondre aux besoins suivants:

Réutilisation : même si on peut utiliser une transformation ad-hoc pour modifier ponctuellement un modèle particulier, la plupart des transformations sont conçues pour être réutilisées sur plusieurs modèles différents. Une transformation est paramétrée par les modèles, et plus généralement par les méta-modèles des modèles qu'elle manipule, et

le contenu concret de ces modèles n'est connu que lorsque la transformation est appliquée.

Composition: On veut être en mesure de réutiliser le code d'une transformation existante en l'appelant depuis une transformation englobante. On doit donc pouvoir, au sein de l'environnement et du langage de transformation, exprimer du code de contrôle, pour articuler différentes transformations entre elles.

Généricité: Certaines transformations peuvent être appliquées indépendamment du degré de précision du modèle auquel elles sont appliquées, voire indépendamment des différences entre versions mineures d'un même métamodèle (uml 1.1 à 1.5 par exemple).

Configuration: D'un projet à un autre, ou pour adapter un logiciel à différentes variantes d'une même plateforme, on a souvent besoin d'appliquer les mêmes transformations en n'adaptant que quelques parties spécifiques; de même, dans le contexte des lignes de produits, une transformation générique sera spécialisée pour un domaine d'application restreint.

Maintenance : À partir du moment où on considère que les transformations sont des produits logiciels qui seront réutilisés, il faut pouvoir les maintenir et les adapter en fonction de l'évolution du domaine et des besoins des utilisateurs.

Il existe d'autres besoins tels que la bidirectionnalité qui rend possible l'application d'une transformation dans les deux sens, c.-à-d. on peut retrouver les modèles d'origine à partir du résultat. Les mises à jour incrémentales, un autre besoin, désignent la possibilité de reporter sur le modèle résultat des changements faits sur le modèle source après l'application de la transformation. Si on ne souhaite pas se limiter à des transformations réversibles, il faut recourir sur des mécanismes de traçabilité des transformations ou calculer et mémoriser les différences entre modèles; cela rejoint des travaux récents d'Alanen et al. [13].

2.3.6 Caractéristiques désirables d'un langage de transformation

D'autre part, Sendall et Kozaczynski [86] ont identifié les caractéristiques désirables d'un langage de transformation de modèles. Selon eux, pour supporter le développement dirigé par les modèles, un tel langage devrait:

- être exécutable
- avoir une implémentation efficace

- être expressif mais pas ambigu aussi bien pour les transformations qui modifient des modèles existants (ajout, modification ou suppression d'éléments) que pour les tranformations qui créent des modèles de toutes pièces
- donner une description précise, concise et claire des transformations pour favoriser la productivité lors du développement:
  - en différenciant clairement la description des règles de sélection dans le modèle source et les règles de production du modèle cible
  - en offrant des constructions graphiques pour les cas où une telle représentation est plus concise et intuitive qu'une notation textuelle
  - en rendant implicites les concepts qu'on peut déduire intuitivement du contexte
- donner le moyen de combiner les transformations par au moins des opérateurs de séquence, de condition, et de répétition, pour construire des transformations composites
- permettre de définir les conditions requises pour qu'une transformation soit exécutée

2.3.7 Langages et Outils

Depuis 2002, suite à une intense recherche (industrielle et universitaire), plusieurs outils et langages de transformation de modèles ont vu le jour.

DB-MAIN

DB-MAIN [32] est un CASE tool (Computer-Aided Software Engineering tool) dédié à l'ingénierie des applications de base de données. Un CASE tool est un programme informatique utilisé dans un processus de développement et qui a pour but de supporter une activité de l'ingénerie du logiciel. DB-MAIN est lui utilisé pour la conception des bases de données ou le reverse engineering. Il est capable de transformer automatiquement un schéma entité-relation vers un modèle relationnel. Il peut également générer à partir d'un schéma du code SQL pour différents types de SGBD.

QVT

Query/View/Transformation, aussi connu sous le nom de QVT, a été proposé par l'OMG. Il permet d'exprimer des transformations de modèles. Le standard QVT définit un ensemble de langages permettant d'exprimer des transformations de modèle-vers-modèle:

- QVT-Relational est un langage déclaratif.

- QVT-Operational est un langage hybride qui propose une structure déclarative à base de règles et permet l'utilisation d'expressions impératives.

- QVT-Core définit la sémantique des concepts déclaratifs.

ATL

ATLAS Transformation Language (ATL), est également un langage de transformation de modèle qui a été inspiré par QVT. C'est une approche hybride mêlant des constructions déclaratives et impératives permettant respectivement de traiter des constructions simples ou complexes. ATL est une approche utilisant des modules pour contrôler les transformations. Il peut être intégré à Eclipse grâce à un plug-in.

GReAT

GReAT, Graph Rewriting and Transformation, est un outil permettant de construire des transformations de modèles. Comme mentionné plus haut, cet outil utilise la théorie des graphes et utilise une syntaxe abstraite (celle des méta-modèles) pour spécifier les modèles sources et cibles sous forme d'arbre. Cet outil permet des transformations complexes qui sont exprimées sous la forme de règles de réécriture de graphes. Les règles sont séquencées et sont définies graphiquement par l'utilisateur.

VIATRA

VIATRA, VIsual Automated model TRAnsformations, propose un framework intégré à Eclipse, dont le but est de suivre tout le processus de développement de transformation de modèles en partant des spécifications jusqu'à la maintenance, en passant par le design, l'exécution et la validation des transformations. Tout comme ATL, VIATRA2 propose une approche hybride (déclarative et impérative), et utilise les transformations de graphes comme GReAT. En revanche, il n'est pas possible d'établir un ordre d'exécution des règles.

ETL

L'Epsilon Transformation Language (ETL) [53] est un langage de transformation hybride, de type model-to-model. Il peut gérer plusieurs modèles sources et plusieurs modèles cibles. Il offre des mécanismes de planification des règles et d'héritage entre règles. Du code externe peut églament être exécuté depuis le corps des règles.

Autres

Les approches décrites précédemment sont des solutions qui marchent correctement. Mais en règle générale, la philosophie derrière ces approches est de spécifier des transformations, qui pourront ensuite être automatisées via un langage de transformation de modèles. Le problème est que les utilisateurs ne sont pas toujours familiers avec les règles de transformation de modèles qui sont parfois très complexes à trouver. Ne fut-ce que travailler avec les méta-modèles constitue déjà une barrière pour l'utilisateur qui doit maîtriser parfaitement le(s) langage(s) avant de pouvoir proposer une règle de transformation. De plus, un méta-modèle n'est pas toujours suffisamment expressif et "cache" des concepts que les utilisateurs pourraient ne pas trouver sans avoir une connaissance en profondeur du langage. Afin de pallier ces problèmes, une nouvelle approche, la transformation de modèle par l'exemple, a été proposée et fait déjà l'objet de recherches intenses.

Beaucoup d'autres outils et langages de transformation de modèles existent. Voici une liste non-exhaustive de divers projets autour de ce domaine: mediniQVT [43], Textual Concrete Syntax (TCS) [47], XText [7], MOLA [49], SiTra [11], Beanbag, UMT, MPS Transformation, Microsoft DSL Tool Transformations, GMT, Fujaba Transformations [48], TXL [25], Stratego [21].

2.4 Programmation par l'exemple

La programmation par l'exemple, également connue sous le nom de programmation par démonstration, est une technique qui permet d'enseigner à un ordinateur de nouveaux comportements en lui montrant des actions à effectuer sur des exemples concrets. Le système "enregistre" la série d'actions effectuée par l'utilisateur et infère un programme qui peut être utilisé sur de nouveaux exemples similaires.

Cette technique est née dans le cadre de la recherche du génie logiciel au milieu des années 80 et apparaît comme un moyen permettant de définir une série d'opérations sans avoir à apprendre un langage de programmation.

Un système PbE (programming by example) reçoit de l'utilisateur des exemples qu'il exécute afin de les évaluer. Ensuite, à partir des résultats, il va pouvoir créer une preuve qui pourra se généraliser pour tous les exemples possibles à générer.

Ces systèmes sont également de plus en plus utilisés comme prototypes au niveau de la recherche mais aussi au niveau industriel. Malgré les efforts de conception et les efforts d'encodage des instructions réalisés par les

développeurs, comment savoir si ces instructions fonctionnent réellement et comment peuvent-ils être sûrs que ce qu'ils codent est vraiment ce que désire le client ?

D'un autre côté, le client, dû à son manque de connaissance et à son manque de technique, a également du mal à suivre le développement du produit, ce qui ne favorise pas la surveillance des exigences ni la communication. Ce manque de compréhension est la cause de nombreuses modifications des spécifications, dues par exemple à l'omission d'une particularité du système, et peut augmenter radicalement les coûts de développement.

Par contre, avec ces prototypes, les utilisateurs (clients) peuvent tester très tôt le produit avec différents exemples et les développeurs peuvent également valider ou corriger certaines méthodes. Ce sont les futurs utilisateurs du programme qui produisent les exemples que ce dernier devra exécuter. Les relations entre les développeurs et les utilisateurs sont renforcées et grâce à ces simulations, les clients peuvent donc suivre plus facilement l'évolution de leur investissement. De plus, ces systèmes permettent aux personnes non initiées à la programmation d'apprendre à utiliser le programme plus rapidement et plus facilement. Dès lors, la programmation par l'exemple peut être considéré également comme un outil d'apprentissage.

Les recherches démontrent que la programmation par l'exemple est une méthode efficace pour rendre l'informatique et le programme plus accessible à plus de personnes. C'est une bonne méthode pratique pour la création de logiciels. Ce paradigme peut également être utilisé pour pouvoir créer des plans de transformation de modèles sans l'aide d'expert en la matière et sans avoir à connaître de façon précise toutes les particularités des modèles et de leur méta-modèle. A l'aide d'exemples de modèles sources et cibles fournis en entrée, le programme devra fournir le plan de transformation idéal qui transformera les modèles sources en modèles cibles.

2.5 Transformation de Modèle par l'exemple

L'idée principale derrière la transformation de modèle par l'exemple est de dériver des programmes de transformation, à l'aide de transformations concrètes que l'on peut trouver dans un ensemble d'exemples de modèles sources et cibles. Dans les travaux existants autour de la transformation de modèle par l'exemple, la plupart des approches utilisent des exemples représentés par des triplets, constitués du modèle source, du modèle cible et des traces de transformation entre les deux modèles. Les exemples sont exprimés dans une syntaxe concrète, généralement définie par l'utilisateur.

La transformation de modèles par l'exemple permet à l'utilisateur de définir des mappings entre les modèles sources et cibles, situés au niveau M1 (fig. 2.3). Cela rend donc la transformation plus "user-friendly" puisque celui-ci n'a pas à connaître la syntaxe du méta-modèle (au niveau M2) qui est moins facile à maîtriser. Le mapping entre ces modèles peut être utilisé pour générer des règles de transformation par l'exemple. Les connaissances de l'utilisateur par rapport aux notations du langage de modélisation sont suffisantes pour la définition de transformation de modèles. De plus, aucune connaissance des détails relatifs à la syntaxe abstraite n'est requise.

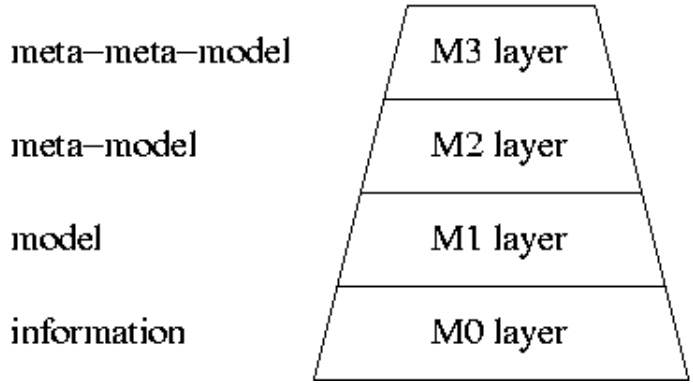

Figure 2.3: *Les différentes couches de modèles*

La première approche de transformation de modèles ayant utilisé des exemples de transformations a été proposée par Varró en 2006 [99]. Il a été inspiré par les transformations de fichier XML à l'aide de règles XSLT, générées automatiquement à partir de documents ou schémas. Son approche a généralisé le processus aux transformations de modèles. Il dérive des règles de façon (semi-)automatique à partir d'un ensemble de modèles sources et cibles. Les règles devaient ensuite être raffinées par l'utilisateur, qui pouvait ajouter des conditions par exemple. Varró utilise la logique inductive pour générer ses règles de transformation. Varró et Balogh [100] ont automatisé cette approche en utilisant la programmation logique inductive.

Wimmer et al. [104] ont utilisé la MTBE et les contraintes existantes dans EMF [89] et GMF [2] pour dériver des règles ATL à partir d'exemples décrits en syntaxe concrète, contrairement à Varró et Balogh qui utilisent des exemples abstraits. Là où Varró et Balogh utilisent des graphes pour dériver les règles de transformation de modèle, Wimmer et al. utilisent une approche à base d'objets (une classe, un attribut). Les approches de Wimmer et al. et de Varró et Balogh utilisent la correspondance sémantique qui unit les modèles pour dériver les règles.

Dolques et al. [84] proposent une approche permettant de s'acquitter du traditionnel triplet d'exemple de transformation contenant un modèle source, un modèle transformé et les liens entre les éléments source et les éléments correspondants transformées. La création des liens est faite à la main et est fastidieuse. Mais dans de nombreux cas, les liens peuvent être déduits de l'examen de la source et de la cible en utilisant des propriétés pertinentes comme les identifiants. C'est à partir des liens que les règles de transformation sont générées. Étant donné le fait qu'ici il n'y en a pas, les exemples fournis doivent couvrir toutes les possibilités.

Kessentini et al. [50] ont également travaillé sur la transformation de modèle par l'exemple et ont proposé une méthode permettant la dérivation automatique de règles de transformation de diagrammes d'états vers des réseaux de pétri colorés. Ils ont pour cela utilisé les méta-modèles des deux langages. Leur approche consiste en la génération et l'évaluation des règles selon des méta-heuristiques. Ils ont utilisé l'optimisation par essaims particulaires (Particle Swarm Optimization, PSO) et le recuit simulé.

Les approches présentées jusque maintenant sont assez limitées, dans le sens où il n'est pas possible de dériver des règles de transformation testant plus d'une construction dans le modèle source, et produisant plus d'une construction dans le modèle cible.

Wimmer et al. produisent des règles 1-à-1, Kessentini et al. proposent du 1-à-plusieurs, Strommer et Wimmer [92] du plusieurs-à-1. García-Magariño et al. [36] ont proposé une approche permettant de dériver des règles plusieurs-à-plusieurs et permettent également de passer au-delà des limitations des langages de transformation ne permettant pas d'exprimer directement de telles règles. L'approche a été concrétisée avec l'implémentation d'un algorithme capable de générer des règles ATL.

Saada et al. [84] proposent une approche en deux étapes pour générer des règles de transformation. Dans la première étape, à partir d'exemples, des patrons de transformation sont appris au moyen d'une classification des éléments des modèles et d'une classification des liens de transformation. Pour cela, ils utilisent l'analyse de concepts formels. Dans la seconde étape, les patrons de transformation sont analysés et les plus adéquats sont sélectionnés. Ils sont ensuite transformés en règles de transformation qui sont utilisées avec le moteur d'inférence JESS. Ils exécutent les règles sur des exemples et les évaluent. Cette approche permet également de créer des règles de type plusieurs-à-plusieurs.

Malgré le fait que des règles de transformation complexes soient quasi inévitables dans la transformation de modèles, les approches existantes ne

permettent pas de créer de telles règles, c'est-à-dire des règles mettant en relation plusieurs constructions et concepts du langage, testant la non-existence d'une construction ou d'autres propriétés des constructions. Dans la plupart des cas, il faut raffiner à la main les règles générées.

Un autre point concerne l'ordonnancement des règles. Selon les technologies utilisées, l'ordre d'exécution des règles peut être implicite, défini par les règles elles-mêmes (sans intervention de l'humain), ou bien explicite, via du méta-contrôle. Les approches existantes transforment donc des constructions d'un modèle source en constructions d'un modèle cible. Cependant, les morceaux cibles obtenus pourraient être dépendants entre-eux et doivent donc être connectés afin de former un modèle cible cohérent. Le problème vient du fait que les règles ne peuvent pas tester l'existence de constructions dans le modèle cible et, par conséquent, relier les constructions produites ensemble. Pour pallier ce problème, les approches existantes utilisent des espaces de noms identiques pour les modèles cibles et sources pour lier les constructions implicitement.

En dehors de la transformation de modèle par l'exemple, il existe une autre approche appelée la transformation de modèle par démonstration (Model Transformation By Demonstration, MTBD). Au lieu de fournir des exemples de transformation, c'est l'utilisateur qui doit apprendre à l'ordinateur à transformer des modèles en éditant les modèles sources. Le système enregistre les actions de l'utilisateur et génère des patrons de transformation grâce à un moteur d'inférence. Le patron spécifie les pré-conditions et la séquence d'action à effectuer pour réaliser la transformation. Le patron peut alors être réutilisé sur d'autres constructions respectant les pré-conditions. L'inconvénient de cette approche est qu'elle est lente dans sa courbe d'apprentissage et qu'elle est fortement dépendante de l'humain. Cependant, cette approche n'a été testé que sur des fragments et non des modèles sources entiers.

Brosch et al. [22] proposent un outil (Operation Recorder) permettant de spécifier des "opérations composées" telles que celles utilisées dans le refactoring. Cette approche permet à l'utilisateur de s'acquitter de l'apprentissage des meta-modèles et des langages de transformation. L'utilisateur modélise l'opération composée avec un exemple. Cela permet la génération semi-automatique d'une spécification d'opérations composées, qui pourront être utilisées lors de modélisations futures. Langer et al. [60] et Sun et al. [93] proposent, eux aussi, une approche par démonstration. L'approche est similaire à celle proposée par Brosch et al.. Il est à noter que contrairement aux approches par l'exemple, les approches par démonstration n'utilisent pas le triplet définissant une transformation. En effet, les traces ne sont ici pas utilisées. Langer et al. et Sun et al. permettent l'expression de patrons de

transformation complexes.

Aucune des approches mentionnées ci-dessus n'affirme générer des plans de transformation complets et corrects. D'ailleurs, la plupart des approches spécifient que les transformations complexes doivent être prises en charge par l'utilisateur, ou que certaines règles doivent être raffinées, en particulier les règles impliquant des requêtes complexes, du calcul et du comptage d'éléments. Il n'est cependant pas toujours simple pour l'utilisateur de décréter qu'une règle est "mauvaise".

Chapitre 3

La programmation génétique

3.1 Description

Dans ce chapitre, le concept de programmation génétique sera introduit d'un point de vue théorique et général. Nous verrons les différentes phases qui constituent un algorithme génétique tel que celui utilisé dans ce travail, que nous présenterons par la suite. Chacune des phases sera détaillée, afin d'aider le lecteur à voir clairement l'intérêt de la programmation génétique dans le cadre de ce travail. Nous verrons également comment spécifier un problème à résoudre via un algorithme génétique, et plus spécifiquement les paramètres qui doivent être fournis de sorte à converger vers une solution optimale. Enfin, nous terminerons ce chapitre en présentant les différentes applications de la programmation génétique et les limites de celle-ci.

3.2 Théorie

3.2.1 Définition

L'un des principaux défis de l'informatique actuelle, est d'arriver à faire faire ce que l'on souhaite à un ordinateur sans qu'on lui dise comment le faire. Le but de la programmation génétique est de résoudre ce problème en fournissant des outils qui permettront de créer, de manière automatisée, des programmes adaptés à la résolution des problèmes.

La programmation génétique a été inspirée par le mécanisme de la sélection naturelle tel qu'il a été établi par Charles Darwin [30] pour expliquer l'adaptation plus ou moins optimale des organismes à leur milieu. Elle a pour but de trouver, de manière automatique et par approximations successives, des programmes répondant au mieux à une tâche donnée. Afin d'y arriver, la programmation génétique utilise des algorithmes évolutionnistes

dans le but d'optimiser la population de programmes.

Ces algorithmes évolutionnistes sont appelés algorithmes génétiques. Leur but est d'obtenir une solution approchée à un problème d'optimisation, lorsqu'il n'existe pas de méthode exacte (ou que la solution est inconnue) pour le résoudre en un temps raisonnable. Les algorithmes génétiques utilisent la notion de sélection naturelle et l'appliquent à une population de solutions potentielles au problème donné. La solution est approchée par "bonds" successifs, comme dans une procédure de séparation et évaluation, à ceci près que ce sont des formules qui sont recherchées et non plus directement des valeurs.

La programmation génétique permet, dans une certaine mesure, de répondre au rêve de tout informaticien, à savoir créer un programme pour résoudre un problème donné, et ce de manière totalement automatisée.

La programmation génétique est une branche récente mise au goût du jour par John Koza à la fin des années quatre-vingts [56].

Dans la cadre général des techniques évolutionnaires, une population est constituée d'individus, qui correspondent plus ou moins à une solution potentielle. Cette population va évoluer afin de s'adapter à son environnement (l'environnement correspondant au problème posé), comme le ferait une espèce dans le processus darwinien. Cette population va subir les différentes étapes d'un processus évolutif simulé:

- Sélection des individus les mieux adaptés à leur environnement, c'est-à-dire les programmes répondant le mieux au problème posé.
- Croisement des individus, c'est-à-dire des échanges de codes entre deux programmes.
- Mutation aléatoire des individus, c'est-à-dire des ajouts ou des retraits de code dans un programme.

La partie la plus délicate dans les techniques évolutionnaires est l'écriture d'une fonction d'adaptation. Toujours en parallèle avec le processus darwinien, seuls les individus les mieux adaptés à leur environnement survivent. Il faut donc être capable d'écrire une telle fonction qui permettra d'évaluer les individus afin d'identifier les meilleurs éléments. Cette fonction est généralement appelée fonction objectif, fonction fitness ou encore fonction d'adaptation.

Une difficulté de la programmation génétique réside dans la spécification des "briques" de base qui serviront à créer des programmes. Fournir un

ensemble de briques trop petit pourrait empêcher l'algorithme de tomber sur la solution. Par contre, si cet ensemble est trop large, l'algorithme pourrait peiner à tomber sur la solution à cause d'un espace de recherche trop important [34].

3.2.2 Algorithme

Quatre étapes sont nécessaires à la résolution d'un problème:

1. Générer une population initiale au hasard (génération 0), dont les individus sont composés des ingrédients disponibles. Ce sont des solutions de bases du problème.

2. Exécuter et évaluer chaque individu (programme) de la population afin de lui assigner une valeur (fitness) en fonction de son efficacité à résoudre le problème.

   Variante: fusionner les deux premières étapes. On génère une population qui respecte un certain seuil de qualité.

3. Appliquer ces étapes jusqu'à rencontrer le critère d'arrêt:

   a) Sélectionner un ou plusieurs individus de la population avec une probabilité basée sur leur fitness.

   b) Créer de nouveaux individus en appliquant les opérateurs génétiques suivants:

      i) Reproduction/Sélection: Copier l'individu dans la nouvelle population.

      ii) Croisement: Créer de nouveaux individus à partir de la combinaison de deux programmes sélectionnés.

      iii) Mutation: Créer un nouvel individu pour la prochaine population en mutant une partie sélectionnée du programme.

   c) Exécuter chaque programme de la nouvelle population et lui attribuer une valeur de fitness.

4. Une fois le critère d'arrêt rencontré, le meilleur programme de la population est retenu et présenté comme solution du problème ou la solution qui s'en rapproche le plus.

3.2.3 La fonction de fitness

Une des parties les plus importantes et complexes de la programmation génétique est sans nul doute la fonction de fitness. Cette fonction mesure l'adéquation d'un programme pour résoudre le problème.

3.2.4 Les opérateurs

La génétique a mis en évidence l'existence de plusieurs opérations au sein d'un organisme donnant lieu au brassage génétique. Ces opérations interviennent lors de la phase de reproduction lorsque les chromosomes de deux organismes fusionnent. Ces opérations sont imitées par les algorithmes génétiques afin de faire évoluer les populations de solutions de manière progressive.

Les sélections

Pour déterminer quels individus sont plus enclins à obtenir les meilleurs résultats, une sélection est opérée. Ce processus est analogue à un processus de sélection naturelle, les individus les plus adaptés gagnent la compétition de la reproduction tandis que les moins adaptés meurent avant la reproduction, ce qui améliore globalement l'adaptation. Etant donné que la sélection est le résultat d'une intervention humaine ou, du moins, l'application d'un critère défini par l'homme, les algorithmes génétiques devraient donc plutôt être rapprochés de la sélection artificielle telle que la pratiquent les agriculteurs que de la sélection naturelle, qui œuvre "en aveugle".

Les croisements

Lors de cette opération, deux chromosomes s'échangent des parties de leurs chaînes, pour donner de nouveaux chromosomes. Ces enjambements peuvent être simples ou multiples. Dans le premier cas, les deux chromosomes se croisent et s'échangent des portions d'ADN en un seul point. Dans le deuxième cas, il y a plusieurs points de croisement. Pour les algorithmes génétiques, c'est cette opération (le plus souvent sous sa forme simple) qui est prépondérante. Sa probabilité d'apparition lors d'un croisement entre deux chromosomes est un paramètre de l'algorithme génétique. La probabilité d'un croisement est comprise entre 0 et 1.

Les mutations

De façon aléatoire, un gène peut, au sein d'un chromosome être substitué à un autre. De la même manière que pour les enjambements, on définit ici un taux de mutation lors des changements de population qui est généralement compris entre 0,001 et 0,01. Il est nécessaire de choisir pour ce taux une valeur relativement faible de manière à ne pas tomber dans une recherche aléatoire et conserver le principe de sélection et d'évolution. La mutation sert à éviter une convergence prématurée de l'algorithme. Par exemple lors d'une recherche d'extremum la mutation sert à éviter la convergence vers un extremum local.

3.3 Les phases de la programmation génétique

Un programme génétique est découpé en plusieurs phases [17]. Le schéma ci-dessous représente le cycle de fonctionnement d'un tel programme (fig. 3.1). La phase d'initialisation va permettre de construire une population initiale de programmes. Cette population sera constituée d'individus qui seront évalués afin de générer les générations futures. Si aucune solution ne convient dans l'ensemble des individus ou si le critère d'arrêt n'a pas encore été atteint, l'algorithme va effectuer une nouvelle itération. Durant celle-ci, les meilleurs individus vont être sélectionnés pour devenir les géniteurs de la future génération. Différentes techniques (Crossover et Mutation) vont être appliquées sur ceux-ci pour générer les descendants. Les enfants vont remplacer la génération précédente, et le cycle recommence.

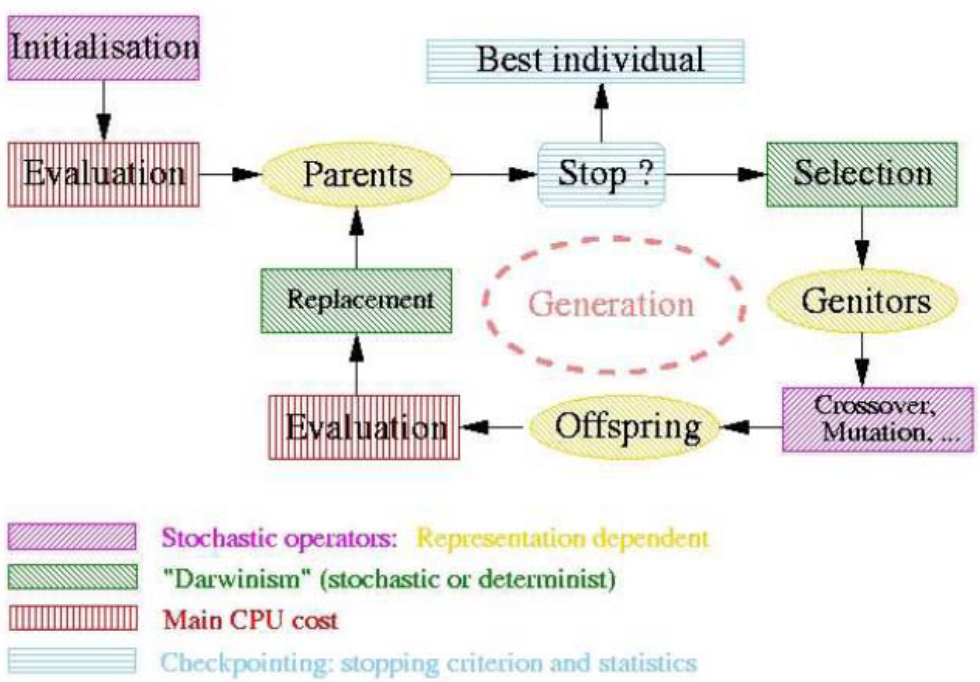

Figure 3.1: *Phases de la programmation génétique*

Génération de la population initiale

La génération de la population initiale est une étape cruciale de l'algorithme. Afin de parcourir un espace de recherche au maximum, il est important de répartir de manière équitable la population initiale sur cet espace. Généralement, un seuil de sélection est prédéfini pour commencer avec une population d'individus qui ne sont pas très mauvais. Cela permet également de limiter

la profondeur de l'arbre de recherche, limitant par la même occasion l'espace de recherche afin d'assurer une convergence du processus.

Evaluation de la qualité d'une solution

La phase d'évaluation va permettre d'évaluer la qualité de chaque individu. La façon de déterminer cette qualité dépend du problème. Une fonction objectif appelée aussi "fitness" aura pour rôle de définir si un individu est bon ou mauvais. Il existe différents types de fitness associé à différents types de problèmes:

- fitness standardisé: La meilleure valeur possible est 0 et toutes les valeurs sont positives.
- fitness normalisé: Similaire à la représentation standardisée, mais les valeurs sont comprises entre 0 et 1.
- fitness ajusté: Similaire à la représentation normalisée mais où le meilleur score possible vaut 1. Souvent calculé par $\frac{1}{1+fitness\,standardisé}$.

La sélection

La phase de sélection est un processus similaire à la sélection naturelle en biologie. Les individus les plus résistants et les plus adaptés à l'environnement vont continuer à évoluer en gagnant la compétition de la reproduction tandis que les individus les plus fragiles vont disparaitre. Dans ce cas-ci, les individus les plus forts sont ceux qui sont les plus aptes à résoudre le problème donné en entrée.

On notera par contre qu'il existe une petite différence entre la sélection naturelle et la sélection artificielle. La sélection naturelle se compose d'une sélection de survie (atteindre les proies, échapper aux prédateurs, gérer les parasites et germes de maladie) et d'une sélection sexuelle (obtenir une descendance). Tandis que la sélection artificielle va sélectionner les individus qui possèdent une ou plusieurs caractéristiques désirés. L'objectif est prédéfini à l'avance. On distingue plusieurs méthodes de sélection:

Sélection par roulette: Pour chaque individu, la probabilité d'être sélectionné est proportionnelle à son adaptation au problème. Le principe de sélection par roulette est le même que celui de la roue de la fortune biaisée (fig. 3.2). Cette roue est une roue de la fortune classique sur laquelle on associe un segment dont la longueur est proportionnelle à la fitness de chaque individu. On effectue ensuite un tirage aléatoire comme pour les roulettes de casinos. Comme on peut le voir sur la figure ci-dessous, les bons individus auront plus de chances d'être sélectionnés étant donné que leur segment prendra plus de place sur la roulette.

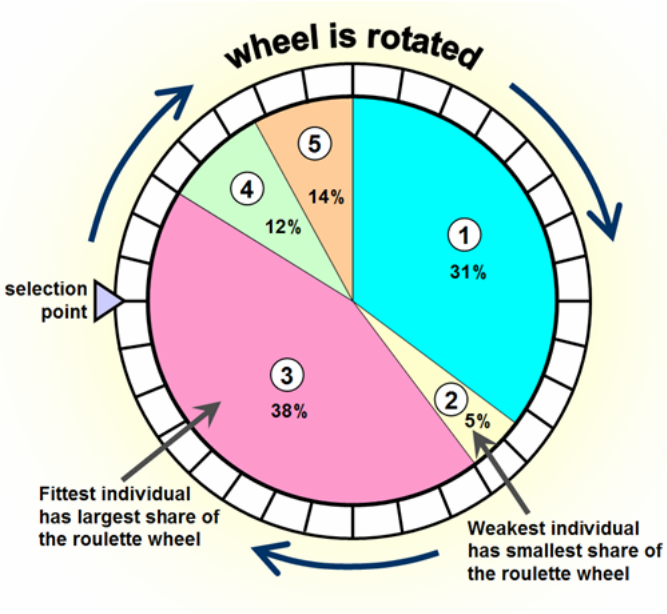

Figure 3.2: *Sélection par roulette*

Sélection par tournoi: La sélection par tournoi consiste à sélectionner N individus au hasard dans la population et à prendre le meilleur parmi ce groupe d'individus sélectionnés. Il faudra alors organiser autant de tournois qu'il y a d'individus à repêcher. Cette technique permet de donner plus ou moins de chance aux individus peu adaptés, puisqu'il se pourrait qu'un groupe soit composé exclusivement d'individus "faibles" et donc le moins faible d'entre-eux remportera le tournoi. Mais avec un nombre élevé de participants, un individu faible sera presque toujours sûr de perdre.

Sélection par rang: La sélection par rang trie d'abord les individus par valeur de fitness. Ensuite, chaque chromosome se voit attribuer un rang en fonction de sa position. Le plus mauvais chromosome aura le

rang 1, le suivant 2, et ainsi de suite jusqu'au meilleur chromosome qui aura le rang N (pour une population de N chromosomes). La sélection d'un chromosome est la même que la sélection par roulette, mais les segments de la roulette sont en relation avec le rang plutôt qu'avec la valeur de l'individu.

Sélection "steady-state": L'idée principale est qu'une grande partie de la population puisse survivre à la prochaine génération. A chaque génération, les meilleurs chromosomes sont sélectionnés pour créer des chromosomes fils. Ensuite les plus mauvais chromosomes sont retirés et remplacés par les nouveaux fils.

Elitisme: Lors de la création d'une nouvelle population, il y a beaucoup de chances que les meilleurs chromosomes soient perdus après les opérations de croisement et de mutation. Pour éviter cela, on utilise la méthode de l'élitisme. Cela consiste à copier une partie des meilleurs chromosomes dans la future génération. Le reste de la population est alors généré selon l'algorithme de reproduction habituel. Les résultats sont largement améliorés puisque cette méthode permet de sauver les meilleurs individus.

Sélection uniforme: La sélection se fait aléatoirement, quelle que soit la valeur de fitness d'un individu. S'il y a N individus dans la population, chaque individu a donc une probabilité $\frac{1}{N}$ d'être sélectionné. Les faibles ont donc autant de chance d'être sélectionnés que les forts.

Croisement

La phase précédente de sélection nous a permis d'obtenir les géniteurs, les individus qui vont pouvoir se reproduire pour générer la nouvelle population. Le croisement va permettre de manipuler les chromosomes de deux géniteurs afin de créer deux enfants. Chacun de ses enfants possédera une partie des chromosomes de chacun de ses parents. En règle générale, un point de croisement sera sélectionné aléatoirement afin de découper le chromosome de chaque parent en deux (fig. 3.3). Le premier exemple montre que le point de croisement sélectionné est le 4ème carré. L'enfant C1 sera généré avec la partie située à gauche du point de croisement fixé sur le chromosome du parent P1 ainsi que la partie située à droite du point de croisement fixé sur le chromosome du parent P2. Inversément, C2 prendra la partie gauche de P2 et la partie droite de P1.

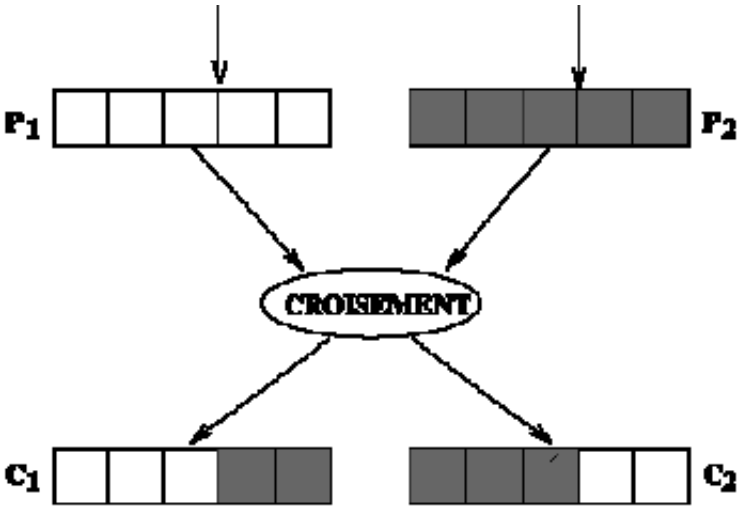

Figure 3.3: *Croisement avec un seul point de croisement*

Il peut également exister des croisements utilisant plusieurs points de croisement (fig. 3.4).

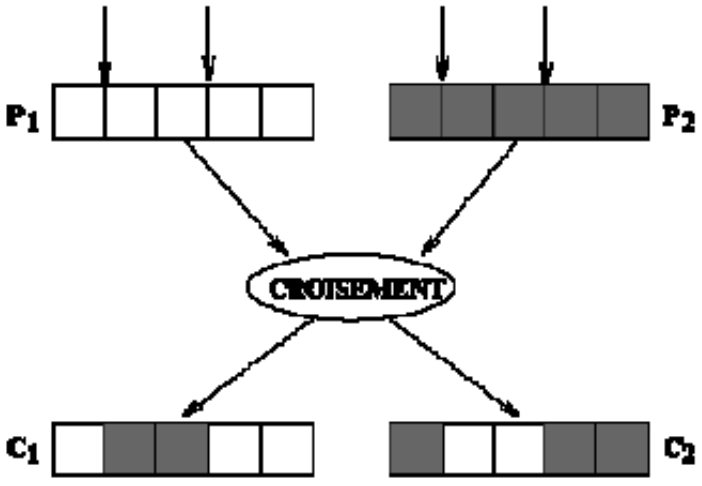

Figure 3.4: *Croisement avec plusieurs points de croisement*

Le choix du nombre de points de croisement diffère selon l'algorithme. Le croisement est une opération dominante dans la programmation génétique et est souvent associé à une probabilité de 85 ou 90%.

Mutation

La mutation contrairement au croisement ne va impliquer qu'un seul parent. Elle permet de modifier un gène d'un chromosome par un autre d'une manière aléatoire. Cette technique peut insérer de nouveaux gènes encore inexistant dans les générations précédentes afin d'augmenter la diversité et évite par la même occasion une convergence prématurée de l'algorithme. Généralement, le taux de mutation est fortement inférieur à celui du croisement. Au plus le taux de mutation sera élevé, au plus le parcours dans l'espace de recherche s'effectuera de manière aléatoire et au moins le principe de sélection et d'évolution sera gardé.

3.4 Spécification d'un problème

La programmation génétique démarre avec l'énonciation des exigences de haut niveau du problème, et finit, ou du moins essaye, de produire un programme résolvant ce problème [55]. C'est bien entendu l'humain qui communique le problème de haut niveau au programme implémentant l'algorithme génétique. Il le communique au travers de cinq étapes préliminaires dans lesquels il devra spécifier:

1. l'ensemble des terminaux, comme par exemple les variables indépendantes du problème, les constantes, etc.
2. l'ensemble des fonctions primitives
3. la fonction d'évaluation du fitness
4. les paramètres permettant de contrôler le déroulement de la génération
5. le critère d'arrêt

Ces cinq étapes préliminaires sont fournies comme paramètres du programme, qui, après avoir terminé son exécution, fournit un programme supposé résoudre le problème (fig. 3.5).

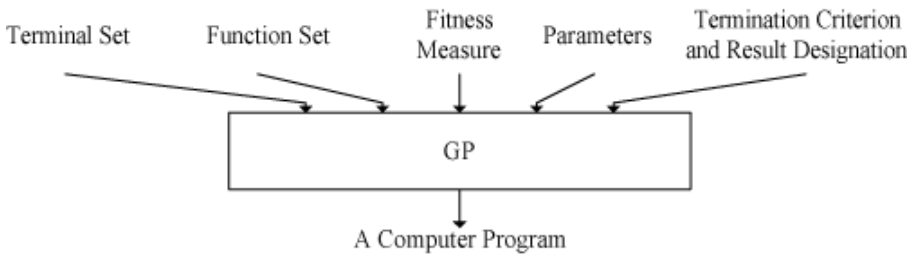

Figure 3.5: *Paramètres à fournir à un algorithme génétique*

Les deux premiers points sont en quelque sorte les ingrédients dont disposera le programme de génération pour créer les programmes qui résoudront le problème.

3.4.1 L'ensemble des fonction et des terminaux

Trouver l'ensemble des fonctions et des terminaux qui seront fournis au programme implémentant l'algorithme génétique est relativement simple. Les fonctions arithmétiques pourraient constituer l'ensemble des fonctions par exemples. L'ensemble des terminaux contiendra les éléments que l'on fournit en entrée du programme génétique, ainsi que des constantes.

Mais l'ensemble de fonctions peut également comporter des fonctions plus spécialisées. Imaginons que l'on souhaite utiliser la programmation génétique pour concevoir un robot qui serait envoyé sur Mars. Il serait alors nécessaire de fournir au programme génétique quelques indications quant aux actions que le robot serait capable de faire telles que "avancer", "prendre une photo", "prélever un échantillon de poussières", etc.

L'avantage de ces ensembles est qu'ils peuvent être réutilisés pour créer d'autres programmes similaires, utilisant ces ingrédients. Ces deux ensembles définissent l'espace de recherche du programme génétique.

3.4.2 La mesure de fitness

La fonction de fitness indique au programme génétique ce dont on a besoin. C'est bien cette mesure qui va permettre de spécifier les exigences de haut niveau du problème au programme. Cette mesure spécifie le but recherché au programme génétique.

La fitness d'un programme peut être mesurée de beaucoup de façons différentes, comme par exemple en calculant le nombre d'erreurs entre sa sortie et la sortie attendue, le temps nécessaire pour arriver dans un certain état, la précision de reconnaissance de patrons, etc.

La fonction objectif est dite multi-objective lorsqu'elle contient plusieurs éléments qui sont en compétition.

3.4.3 Les paramètres de contrôle

Cette étape préliminaire est la spécification des paramètres de contrôle d'une exécution. Parmi ces paramètres, on retrouve par exemple la taille de la population, le nombre de générations, le taux de mutation/croisement, la "taille" d'un programme ou encore la proportion d'individus qui seront sélectionnés dans l'élite. Ces paramètres sont souvent choisis en fonction de la puissance

de calcul dont on dispose, puisque le temps de calcul est proportionnel aux nombres de génération et à la taille de la population.

3.4.4 Le critère d'arrêt

Cette dernière étape préliminaire permet de spécifier le critère qui déterminera si une exécution peut s'arrêter ou non, ainsi que la sélection de la meilleure solution. On trouve fréquemment un nombre de génération comme critère d'arrêt, ou bien une certaine valeur de fitness. Une autre façon de procéder consiste à stopper manuellement l'exécution après avoir vu qu'un certain résultat était atteint. Ensuite, le meilleur résultat est présenté à l'utilisateur.

3.5 Applications de la programmation génétique

Il y a de nombreuses applications de la programmation génétique [17]:

- La "programmation de l'inprogrammable" comportant la création automatique des programmes machine pour les dispositifs de calcul peu usuels tels que les automates cellulaires, systèmes de multi-agent, systèmes parallèles, colonies de fourmis, l'intelligence d'essaim, ...
- Les "Nouvelles inventions commercialement utilisables" (CUNI) [58] utilisant la programmation génétique comme "machine d'invention" automatisée pour créer de nouvelles inventions commercialement utilisables.
- Traitement d'images satellites [29], détection d'objets [106],...
- Prédiction de séries temporelles [62], génération d'arbres de décisions [54], datamining [35], ...
- Classification de segments d 'ADN [40], de protéines [57],...
- Et d'autres [55].

Cependant, aucune approche de transformation de modèle n'utilise la programmation génétique hormis celle que nous présentons.

3.6 Les limites de la programmation génétiques

Comme toute chose, la programmation génétique à ses propres limites. Tout d'abord, cette technique peut être très coûteuse en temps de calcul en comparaison à d'autres heuristiques. En effet, la fonction de fitness est appelée très fréquemment. De plus, il n'est pas toujours certain que l'on trouvera la solution optimale, même après un grand nombre de génération. On pourra dire que l'on est sûr d'avoir approché cette solution optimale, mais peut

être qu'une autre configuration donnera de meilleurs résultats. D'ailleurs, il n'est pas toujours aisé de configurer cette approche. Quelle taille de population doit-on utiliser ? Combien de générations doit-on faire ? Quels taux de croisement et de mutation doit-on utiliser ? Le succès dépend souvent de ces différents paramètres et il faut très souvent plusieurs essais avant de trouver la bonne configuration. Aussi, la fonction d'évaluation est critique et doit être parfaite. Enfin, un dernier problème concerne ce que l'on appelle les optima locaux. Lors d'une exécution, il se peut que certains individus de la population, à un certain moment, soient dominants au sein de la population. Ils vont alors devenir majoritaires, et il se peut que toute la population converge vers eux et s'écarte alors des "bons" individus mais trop minoritaires.

Chapitre 4

Dérivation de règles selon un processus évolutionnaire

4.1 Description

Ce chapitre constitue le centre de ce mémoire. Dans celui-ci, nous présenterons tout d'abord comment la théorie expliquée dans les chapitres précédents a été utilisée dans le cadre de ce travail. Ainsi, nous verrons l'intérêt d'une telle approche, la façon dont les méta-modèles et modèles sont encodés et spécifié à un algorithme génétique, spécialement façonné pour répondre à nos besoins. Une présentation de la solution initiale sera ensuite proposée. Nous verrons notamment comment les règles sont encodées, comment fonctionne l'algorithme génétique et comment les programmes sont évalués. Au travers de cette présentation de l'approche initiale, nous pourrons découvrir ses limitations par rapport aux transformations de modèles complexes. A partir de ses limitations, deux approches seront proposées en vue de les éliminer. Cette partie est scindée en deux, l'approche RAC (Rule And Control) et l'approche RTC (Rule Then Control). Ces deux approches proposent toutes deux des améliorations qui seront détaillées et illustrées par des exemples.

4.2 Problème général

Notre but est de dériver des règles de transformation parfaites, ne devant pas être retravaillées par l'utilisateur. Cependant, atteindre une telle qualité n'est pas chose aisée et de nombreuses approches ont été proposées par la communauté Machine Learning. Parmi les méthodes d'apprentissage disponibles, la programmation génétique semblait la plus adaptée à nos besoins puisque, selon sa définition, c'est une méthodologie permettant de trouver des programmes réalisant une tâche prédéfinie. Les programmes que nous souhaitons obtenir sont des programmes de transformation de modèles,

contenant des règles de transformation exécutables. Ces règles analysent les éléments d'un modèle source et produisent les éléments correspondants exprimés dans le langage du modèle cible. Les programmes sont donc des ensembles de règles transformant un modèle source en un modèle cible.

Le résultat d'un algorithme génétique est un programme pouvant résoudre un problème que l'on a préalablement spécifié en entrée. Le problème est dans notre cas une transformation d'un modèle source en un modèle cible. La spécification du problème consiste donc à fournir à l'algorithme une paire d'exemples de modèles que l'on notera <Ms, Mc>; Ms pour modèle source et Mc pour modèle cible. Les modèles sources et cibles sont tous deux cohérents par rapport à un méta-modèle source (fig. 4.1) et méta-modèle cible (fig. 4.2) préalablement élaboré:

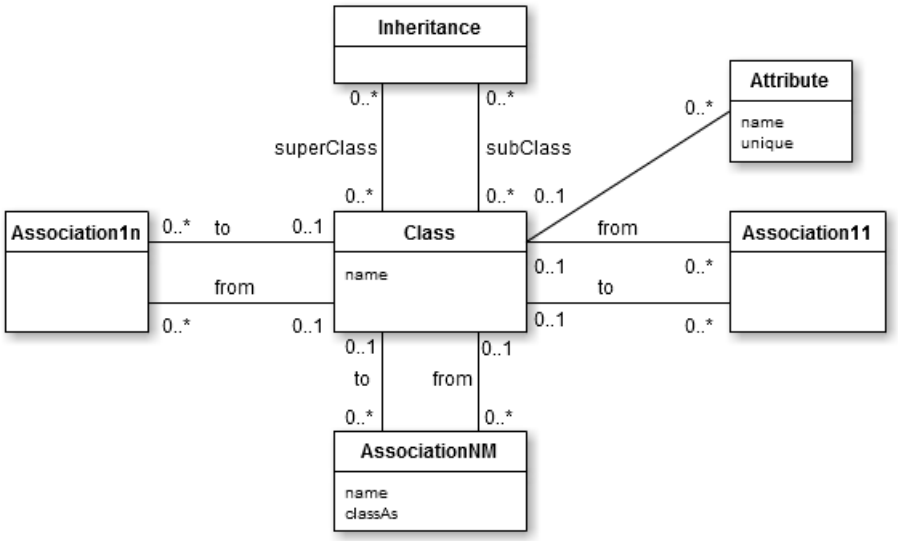

Figure 4.1: *Méta-Modèle du diagramme de classe*

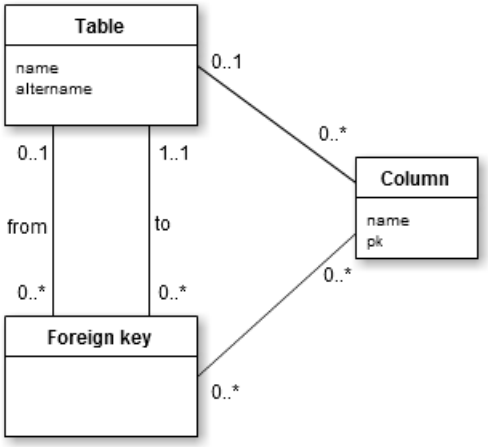

Figure 4.2: *Méta-Modèle du schéma relationnel*

Grâce à ces deux méta-modèles, il est possible de réaliser un modèle de diagramme de classe (fig. 4.3) et un modèle de schéma relationnel (fig. 4.4).

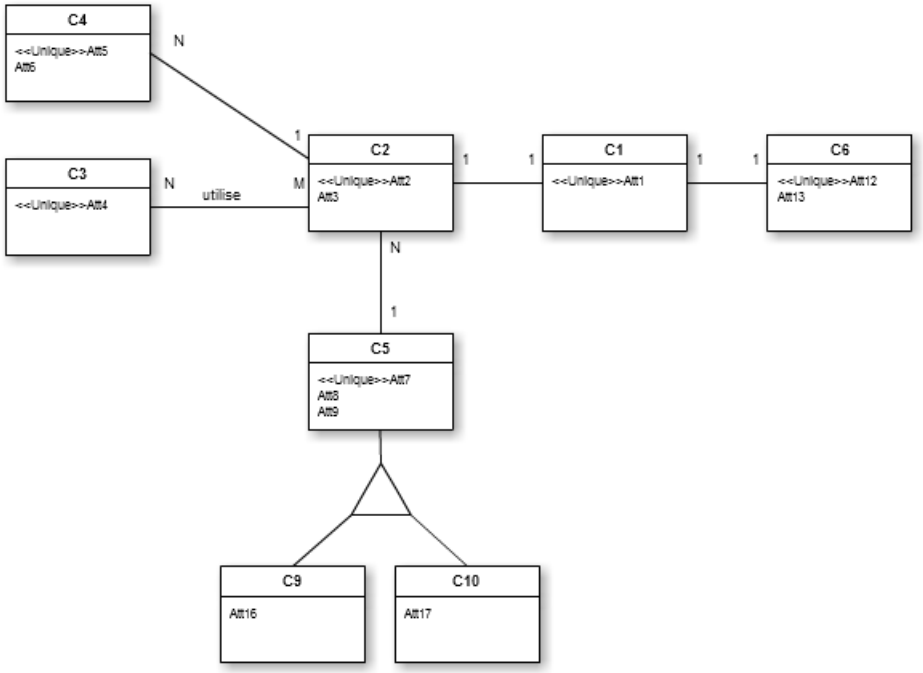

Figure 4.3: *Exemple de modèle de diagramme de classe*

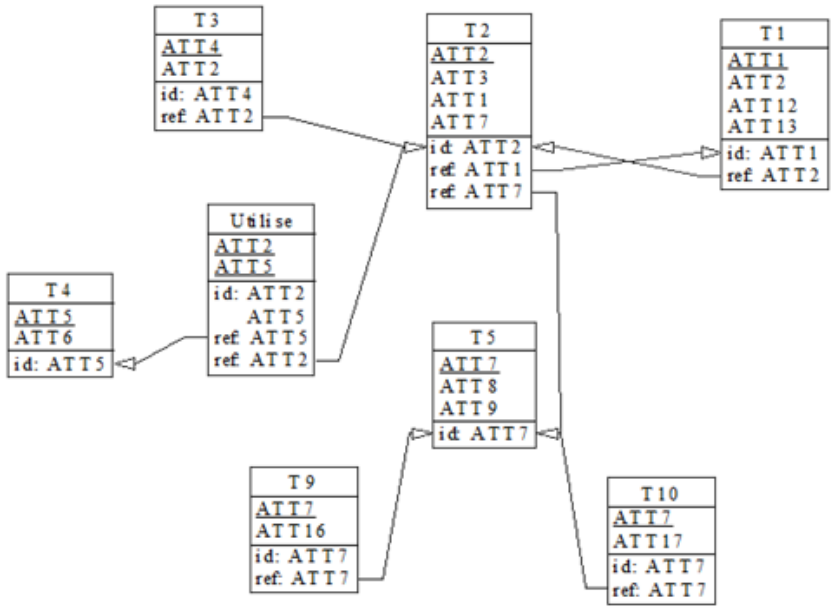

Figure 4.4: *Exemple de modèle de schéma relationnel*

Ces modèles sont ensuite encodés de sorte à pouvoir être traités par notre programme. Exemple d'encodage d'un modèle de diagramme de classes:

```
(assert (class(name classe1)))
(assert (class(name classe2)))
(assert (class(name classe3)))
(assert (attribute(name att1)(class classe1)(unico 1)))
(assert (attribute(name att2)(class classe2)(unico 1)))
(assert (attribute(name att3)(class classe3)(unico 1)))
(assert (inheritance(class classe2)(superclass classe1)))
(assert (inheritance(class classe3)(superclass classe1)))
```

L'algorithme génétique utilisé (fig. 4.5) peut se résumer comme suit:

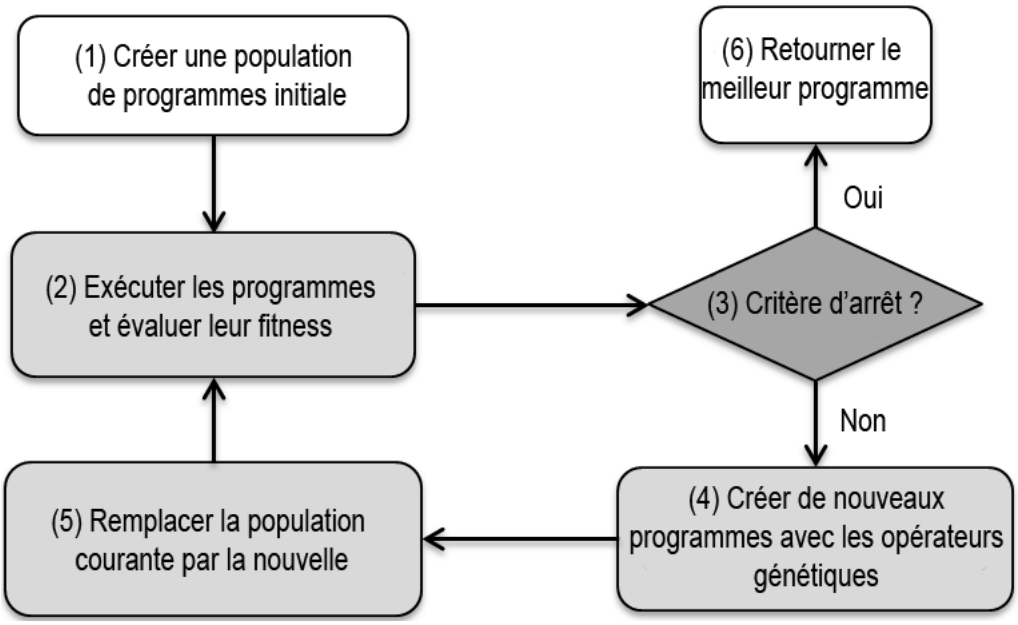

Figure 4.5: *Algorithme génétique*

La première étape consiste à créer une population initiale de programmes de transformation de modèles. Ceux-ci sont composés de règles générées aléatoirement et comporte un nombre de règles étant compris entre deux bornes spécifiées par l'utilisateur.

Les programmes sont alors exécutés et évalués. Les programmes reçoivent en entrée le modèle source et produisent un modèle cible. Le modèle résultant est alors comparé avec le modèle cible attendu, et une valeur de fitness est attribuée à chaque programme.

Après avoir évalué chaque individu/programme de la population, on vérifie si l'on a atteint le critère d'arrêt. Il existe plusieurs sortes de critères d'arrêt. On peut imaginer arrêter le processus après un certain nombre d'itérations, appelées ici générations, ou bien une fois qu'un programme atteint une valeur de fitness maximale. Le critère d'arrêt sera choisi en fonction du résultat souhaité. Si ce critère d'arrêt est rencontré, l'algorithme retourne le meilleur programme, c'est-à-dire le programme transformant le mieux le modèle source en modèle cible.

En revanche, si ce n'est pas encore le moment de s'arrêter, l'algorithme crée la future population de programmes en utilisant des opérateurs génétiques:

le croisement et la mutation. La population ayant servi à créer la nouvelle est alors remplacée par cette dernière. Il faut ensuite réévaluer les individus et le cycle recommence jusqu'à atteindre le critère d'arrêt.

Durant ce cycle, les individus évoluent progressivement grâce aux opérateurs génétiques. Les programmes deviennent de plus en plus adaptés à la résolution du problème, comme l'évolution a rendu les différentes espèces de plus en plus adaptées à leur environnement.

La population dont il est question dans les algorithmes génétiques est la plupart du temps composée de programmes. Dans notre approche, un programme p correspond à un programme de transformation de modèles et est constitué d'un ensemble de règles de transformation, chacune produisant un ou plusieurs fragments du modèle cible.

4.3 Approche initiale

Dans cette section, nous décrivons la solution telle que nous l'avons trouvée. Cela nous permettra de voir ses limitations par rapport à certains problèmes de transformation et comment elles ont pu être éliminées. La solution actuelle a été évaluée selon trois problèmes de transformation, couvrant des caractéristiques importantes dans la modélisation : la structure, la dynamique, et l'imbrication. L'aspect structurel a été testé avec le problème de la transformation de diagrammes de classe en schémas relationnel. La dynamique a été testée par la transformation de diagrammes de séquences simples en diagrammes d'états. Les contraintes temporelles entre les constructions doivent être préservées dans cette transformation. L'imbrication a été testée avec des diagrammes de séquence imbriqués, à transformer en diagrammes d'états. L'imbrication implique que certaines transformations doivent être réalisées avant d'autres. L'approche a obtenu des résultats corrects pour les trois problèmes. Cependant, les méta-modèles utilisés pour représenter ces différents modèles/langages ont été simplifiés et mettent de côté certains problèmes courants de ces types de modèles. La solution actuelle peut atteindre des scores de 100% mais uniquement sur des exemples de tailles réduites et "parfaits". Par exemple, toutes les classes d'un diagramme de classe possèdent un identifiant. C'est une hypothèse un peu forte et c'est loin d'être le cas dans la réalité.

4.3.1 Encodage des règles dans la situation initiale

Un programme P correspond à un programme de transformation de modèles et est constitué d'un ensemble de règles de transformation, chacune produisant un ou plusieurs fragments du modèle cible:

$p = r1, r2, ..., rn$

Une règle, quant à elle, correspond à une paire $ri = (SP, TP)$ où SP est le patron qu'il faut trouver dans le modèle source (source pattern), et où TP correspond à la partie à créer (target pattern).

Un SP est une paire (SGC, G) dans laquelle SGC est un ensemble de constructions génériques et ou G est une garde. Une construction générique est la spécification d'une instance d'un type de construction qui doit être matchée avec des constructions concrètes du modèle source. Cet ensemble peut contenir plusieurs éléments du même type de construction, comme deux classes par exemple, ou plusieurs éléments de type différents. Voici un exemple:

Source pattern:

Class C; Attribute A; Association S // Éléments sources génériques

A.class = C. name // Join condition
S.classFrom = C. name

(and(S.maxCardFrom < 1) (S.maxCardTo > 1)) // Garde

Target pattern:

Table T; Column O // Eléments cibles génériques

T.name:= C. name // liens
O.name:= A. name

O.table = T. name // Join statement

Dans cet exemple, $SGC = C, A, S$ où C, A et S représentent respectivement une classe, un attribut et une association. Chaque construction générique dispose des propriétés de son type de constructeur, défini dans le méta-modèle. Lorsqu'elles matchent avec des constructions concrètes, les propriétés sont instanciées avec les valeurs de ces dernières. Par exemple, un

attribut A possède un nom et le nom de la classe à laquelle il appartient. Il est possible, pendant l'exécution, d'accéder aux propriétés des constructions. Le nom de la classe d'un attribut est ce que l'on appelle une Join Property. C'est un moyen de relier l'attribut à la classe à laquelle il appartient. La garde G contient les Join Conditions. Les Join Properties servent à définir l'ensemble des Join Conditions qui permettent de spécifier un source pattern comme un fragment de modèle, c'est-à-dire un ensemble de constructions en relation selon le méta-modèle. Dans l'exemple ci-dessus la Join Condition $A.class = C.name$ stipule que A devrait être un attribut de la classe C tandis que $S.classFrom = C.name$ restreint le pattern aux seules classes qui sont à l'origine de l'association. Les valeurs des propriétés des éléments du source pattern sont rassemblées en un ensemble de terminaux (Tl). Tl est l'union des propriétés des constructions dans SGC et un ensemble de constantes C. Dans l'exemple, les propriétés sont *C.name, A .name, A.class, S.classFrom, S.classTo, S.MaxCardFr, S.MaxCardTo, etc.* Étant donné que les propriétés sont des nombres ou des chaînes de caractères, des constantes telles que $\{0, 1, Empty, True, False, ...\}$ sont ajoutées à l'ensemble des terminaux. Cet ensemble sera utilisé plus tard dans l'autre partie de la règle: le target pattern.

Le target pattern est, quant à lui, un triplet (TGC, B, J) ou TGC, B et J représentent respectivement un ensemble de constructions génériques cibles, un ensemble de liaisons et un ensemble de jointures. Une construction générique cible spécifie une construction concrète à créer dans le modèle cible quand une règle est tirée. Dans l'exemple, deux constructions cibles sont créées : une table T et une colonne O. L'ensemble des liaisons B détermine comment assigner les valeurs des propriétés des constructions créées à partir de l'ensemble des terminaux Tl. Dans l'exemple, la table et la colonne auront respectivement le même nom que la classe et l'attribut sélectionnés. Enfin, les jointures J permettent de connecter les constructions créées afin de former un fragment du modèle cible. Dans l'exemple, La colonne O est assignée à la table T. Ces jointures doivent être conformes au méta-modèle.

4.3.2 Création des programmes

La création de la population initiale est nécessaire au fonctionnement de l'algorithme génétique. Chaque individu de cette population est un ensemble de règles. Chaque ensemble de règles doit être syntaxiquement correct par rapport à JESS et doit être cohérent avec les méta-modèles sources et cibles. Dès lors, les règles doivent décrire des patrons (concepts) cibles et sources valides. Dans la population initiale, un nombre de programmes maximum est déterminé: Nrs. Le nombre de règles par programmes est choisi aléatoirement dans un intervalle donné en paramètre. Pour chaque règle, une combinaison aléatoire d'un fragment de modèle source et d'un fragment

de modèle cible est effectuée. Ces fragments sont des briques qui sont des ensembles de constructions génériques et qui représentent des concepts du méta-modèle, tels que la relation entre un attribut et sa classe, ou la relation d'héritage entre deux classes. En effet, un attribut ne peut exister sans la classe à laquelle il appartient et une relation d'héritage implique forcément deux classes. Ces briques sont définies par l'utilisateur et couvre l'entièreté des possibilités offertes par le méta-modèle. L'existence de ces briques ne dépend pas de l'existence d'autres constructions. La détermination de ces briques ne repose que sur le méta-modèle et pas sur la transformation des modèles. Les règles sont créées en combinant une brique du modèle source qui formera la partie conditionnelle de la règle et une brique du modèle cible qui formera la partie action de la règle. Les Join Conditions sont calculées et ajoutées à la partie conditionnelle et les valeurs des propriétés des constructions impliquées dans la partie droite sont déterminées à partir de l'ensemble des terminaux de la partie gauche. Les valeurs sont assignées aux propriétés en fonction de leur type.

4.3.3 Opérateurs

Afin de faire évoluer les populations de programmes, l'approche utilise des opérateurs génétiques tels que le croisement et la mutation. Bien entendu, ces opérateurs garantissent la validité syntaxique et sémantique des programmes touchés. Avant d'appliquer les opérateurs, une sélection est effectuée dans la population afin de savoir quels individus seront touchés. C'est le principe de la roulette qui est utilisé, donnant plus de chance d'évoluer aux programmes ayant un bon score de fitness, tout en laissant une petite chance aux moins bons. Un opérateur d'élitisme est également utilisé. Celui-ci sélectionne les n meilleurs programmes et les ajoute directement, sans les manipuler, dans la génération suivante.

Croisement

L'opérateur de croisement permet de créer de nouvelles règles au sein d'un programme en mélangeant le matériel génétique de règles sélectionnées. Deux parents sont sélectionnés et sont croisés pour former deux "enfants". Les parents sont découpés en deux sous-ensembles selon un point de découpage choisi au hasard pour chaque parent. Ensuite, chaque parent échange ses sous-ensembles de règles avec l'autre, ce qui donne lieu à deux enfants. Étant donné que les parents sont syntaxiquement et sémantiquement corrects (voir création population initiale et début section), les enfants obtenus le sont également.

La figure (fig. 4.6) illustre un exemple de croisement:

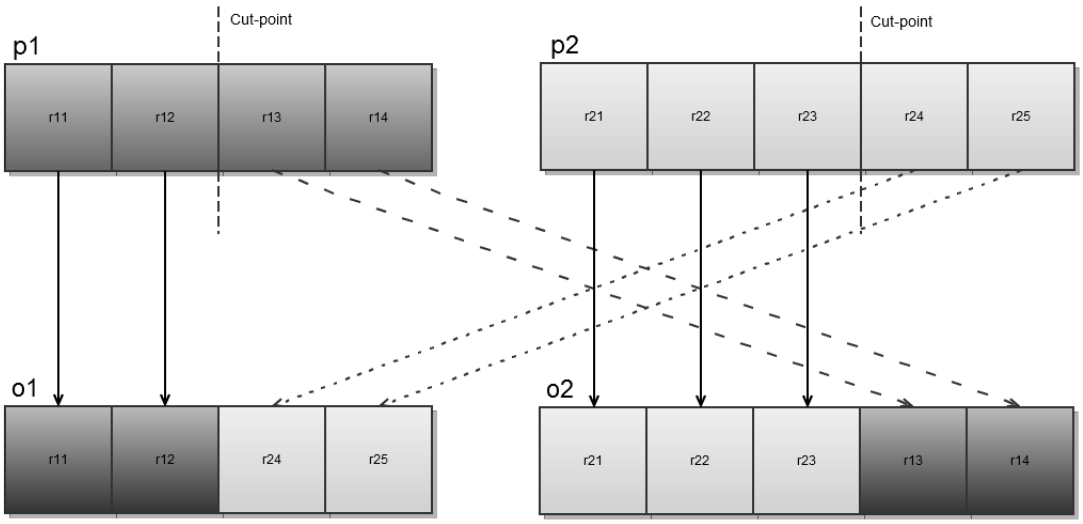

Figure 4.6: *Croisement entre 2 programmes avec un point de découpage aléatoire*

Mutation

Les mutations ont pour but d'introduire du nouveau matériel génétique dans les règles ou d'en supprimer. Plusieurs stratégies ont été définies et touchent soit le programme soit les règles. Les mutations touchant le programme consiste simplement en l'ajout d'une règle aléatoire, ou en la suppression aléatoire d'une règle, tout en veillant à ne pas supprimer la dernière règle d'un programme. Au niveau des règles, il existe également deux stratégies. L'une d'elle consiste à recréer le RHS d'une règle en le remplaçant par une autre RHS créé aléatoirement. L'autre stratégie change les valeurs des propriétés des constructions dans le RHS.

4.3.4 Evaluation des programmes

L'évaluation est le processus qui permet d'évaluer et d'attribuer une valeur de fitness à chaque individu à chaque génération. Ce processus se passe en deux étapes:

1. l'exécution des programmes sur l'exemple de modèle source passé en entrée.

2. la comparaison de ce qui a été produit durant l'exécution avec l'exemple de modèle cible passé en entrée.

Afin d'exécuter les programmes, ceux-ci doivent être préalablement traduits en langage JESS. Les méta-modèles sont décrits sous la forme de templates

de faits et les modèles sont encodés sous la forme de faits, qui sont conformes aux templates de leur méta-modèle. Voici un exemple de règle écrite en JESS:

```
(defrule RuleListing 1
 (class (name ?C1))
 (attribute (name ?A1) (class ?A2))
 (association (maxCardFrom ?S1)(maxCardTo ?S2) (classFrom ?S3))
 (test (and (and (eq ?A2 ?C1) (eq ?S3 ?C1))
 (and (< ?S1 1)(> ?S2 1 ))))
 =>
 (assert (table (name ?C1)))
 (assert (column (name ?A1)(table ?C1)))
)
```

Il est possible que certains faits soient créés plusieurs fois dans l'exécution par différentes règles. Cependant, étant donné que la création consiste en l'assertion du même fait, il est à chaque fois écrasé.

Dans la programmation génétique, la fonction objectif mesure la différence ou le taux d'erreur entre le résultat attendu et ce qui a été produit. Ici, on mesure la différence entre le modèle cible passé en entrée et ce qui a été produit par les règles de transformation. Étant donné que les modèles cibles sont encodés sous la forme d'une liste de faits, et que l'exécution des programmes produit une liste de faits, il est possible de les comparer.

Nous avons donc au départ une paire (Ms, Mc). La mesure de la fitness d'un programme se fait grâce à une fonction $f(p(Ms), Mc)$ qui mesure la ressemblance entre $p(Ms)$, l'application du programme créé sur le modèle source, et Mc, le modèle cible.

La fonction f calcule les moyennes pondérées de l'exactitude de chaque type de constructions $t \in T_{Mc}$. Les constructions ont la même importance, on ne prend pas en compte leur fréquence.
Formellement:

$$f(p(Ms), Mc) = \sum_{t \in T_{Mc}} \frac{ft(p(Ms), Mc)}{|T_{Mc}|}$$

ft est défini comme la somme pondérée des pourcentages de constructions produites durant l'exécution qui correspondent parfaitement (fm), partiellement (pm), ou pas du tout (nm) avec celles du modèle cible:

$$ft(p(Ms), Mc) = \alpha fm + \beta pm + \gamma nm \qquad \alpha + \beta + \gamma = 1$$

Pour chaque construction de type $t \in T_{Mc}$, on détermine si elle correspond totalement à une des constructions produites, c'est-à-dire qu'il existe,

dans le modèle produit, une construction du même type avec les mêmes valeurs de propriétés. Fm est le pourcentage des constructions de Mc qui correspondent parfaitement avec celles du modèle produit. Pour les constructions du modèle cible qui ne correspondaient pas parfaitement, on détermine, dans un deuxième temps, si elles correspondent partiellement. Une construction correspond partiellement s'il existe, dans le modèle produit, une construction du même type qui n'a pas été sélectionnée dans l'étape précédente. Pm est donc égal au pourcentage de constructions du modèle cible qui correspondent partiellement. Nm est le pourcentage des constructions de Mc qui n'ont pas été reprises dans les deux catégories précédentes.

Il serait intéressant de bénéficier d'une mesure de similarité précise pour les correspondances partielles. Cependant, en plus des contraintes liées à la puissance de calcul, le but est de donner un avantage aux règles qui génèrent des constructions du type désiré en espérant que, grâce aux mutations, elles s'améliorent dans les générations suivantes.

Les coefficients α, β et γ ont chacun un impact différent sur le processus de dérivation durant l'évolution. α est, la plupart du temps, la plus grande des trois valeurs (typiquement 0.6). Elle est utilisée pour favoriser les règles qui produisent correctement les constructions attendues. β est une valeur moyenne (0.3) qui permet de donner une chance au règle produisant le bon type de constructions et aide à converger vers la solution optimale. Enfin, γ doit être une petite valeur (0.1). L'idée de donner un petit poids aux mauvaises constructions semble contre-intuitif. Cependant, l'expérience a montré que cela promeut la diversité, en particulier durant les premières générations, ce qui permet de ne pas tomber dans un optimum local trop tôt.

Le calcul de la correction d'une transformation affirme si les constructions du modèle cible sont présentes dans le modèle produit. Cependant, une bonne solution pourrait inclure les bonnes règles qui génèrent les bonnes constructions, mais elle pourrait aussi inclure des règles redondantes ou des règles qui génèrent des constructions non nécessaires. Afin de traiter ce problème, la taille du programme entre en considération. La taille n'intervient pas dans le calcul, mais dans la sélection de la meilleure solution. De ce fait, même si une solution optimale est trouvée en terme de correction, le processus d'évolution continue de chercher des solutions optimales mais avec moins de règles.

4.4 Limites et critiques de l'approche initiale

La solution initiale peut atteindre des scores de 100% mais uniquement sur des exemples de tailles réduites et "parfaits". Par exemple, toutes les classes d'un diagramme de classe possèdent un identifiant. C'est une hypothèse un peu forte et c'est loin d'être le cas dans la réalité. Ce détail peut paraître anecdotique, mais la solution actuelle ne serait pas capable de transformer des modèles dont des identifiants seraient absents.

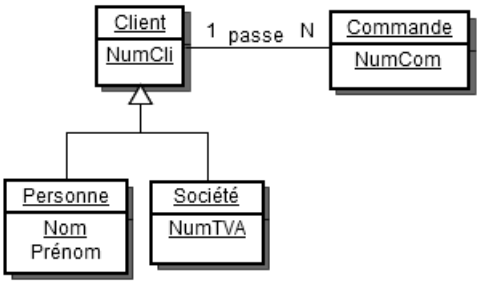

Figure 4.7: *Exemple de diagramme de classe*

Cet exemple de diagramme de classe (fig. 4.7), modélisant le fait qu'un client, de type société ou personne physique, peut passer une ou plusieurs commandes est relativement simple à transformer en un schéma relationnel de même sémantique (fig. 4.8). Les classes doivent être transformées en tables, les attributs en colonnes. Bien entendu, les attributs "unique" (soulignés) sont transformés en clés primaires. Les relations un-à-plusieurs (1-N) sont transformées en utilisant une clé référentielle du coté "plusieurs" (N) vers le coté "un" (1). On procède de la même façon pour la relation d'héritage, puisque les tables associées aux sous-classes se voient attribuer une clé référentielle vers la table associée à leur super-classe.

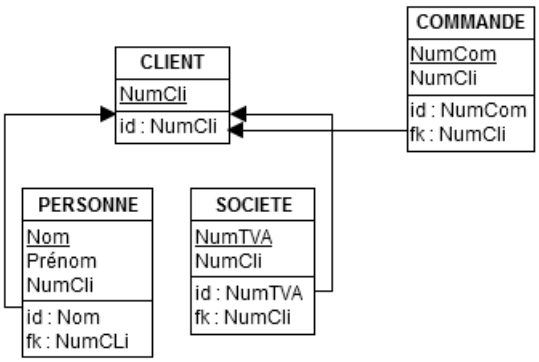

Figure 4.8: *Exemple équivalent à la figure 4.7 en schéma relationnel*

Cet exemple est donc parfaitement gérable par la solution initiale, qui parviendra à trouver une solution en quelques dizaines de générations seulement. On peut également remarquer que cet exemple n'a pas nécessairement besoin de contrôle. Comme déjà expliqué plus haut, les modèles sont représentés sous forme d'une liste de faits. L'important est qu'il soient tous présents à la fin, peu importe l'ordre dans lequel ils sont insérés dans la mémoire. Dans cet exemple, il n'est donc pas important de créer les tables avant les colonnes et les clés étrangères.

Il est intéressant de remarquer que ce schéma est loin de couvrir tous les aspects qu'offre le méta-modèle du diagramme de classe. En effet, il n'y a pas d'association plusieurs-à-plusieurs (M-N) ou un-à-un (1-1). Les classes ont chacune un identifiant, mais il est possible qu'une classe n'en ait pas, comme les sous-classes par exemple, héritant ainsi de l'identifiant de leur super-classe. Cela constitue une pratique assez répandue.

Modifions donc légèrement le premier schéma (fig. 4.7):

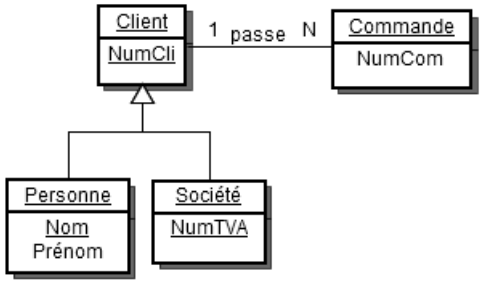

Figure 4.9: *2ème exemple de diagramme de classe*

La modification est mineure et porte sur la classe "Commande" (fig. 4.9). L'attribut "NumCom" a perdu son statut d'identifiant. Il peut en effet arriver qu'une classe n'ait pas d'identifiant.

La figure suivante (fig. 4.10) modélise la même chose en schéma relationnel.

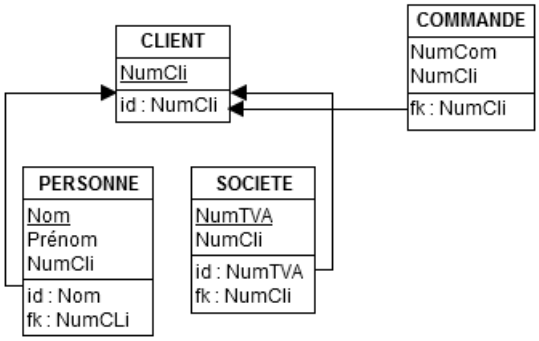

Figure 4.10: *Exemple équivalent en schéma relationnel*

Cette modification n'a pas d'impact sur la transformation et sur sa complexité. Puisque la classe "Commande" n'a pas d'identifiant, la table résultante n'aura pas de clé primaire.

Faisons encore évoluer le diagramme de classe précédent (fig. 4.9):

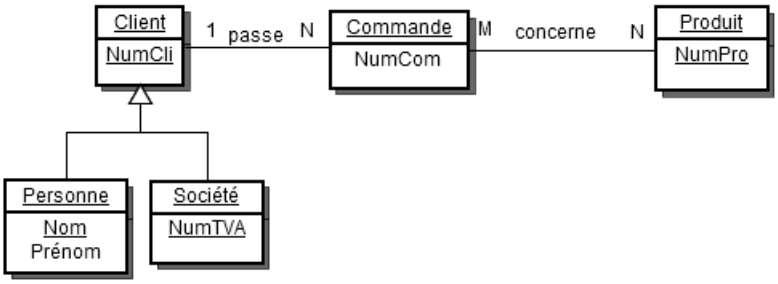

Figure 4.11: *3ème exemple de diagramme de classe*

Ce diagramme de classe introduit le fait que des commandes concernent des produits (fig. 4.11). On a rajouté une relation "concerne" de type plusieurs-à-plusieurs (M-N). Ce type d'association se traduit simplement par l'ajout d'une table intermédiaire, munie de deux clés étrangères (et primaires !) vers les deux tables impliquées dans l'association dans le schéma relationnel. Le schéma relationnel résultant est représenté par la figure 4.12.

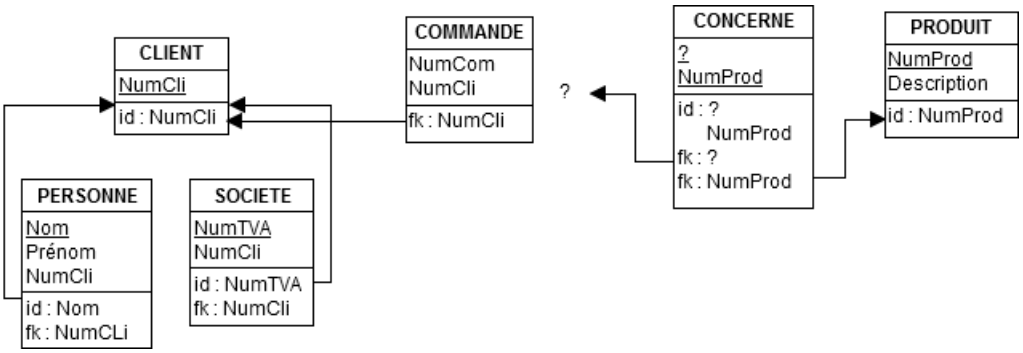

Figure 4.12: *Exemple équivalent en schéma relationnel*

Cette transformation, relativement simple, n'est malheureusement pas faisable avec la solution initiale. En effet, le fait que la table "COMMANDE" ne contienne pas de clé primaire pose un problème puisque celle-ci est impliquée dans une association N-M. La table "CONCERNE" devrait contenir une clé primaire et référentielle vers la table "COMMANDE", mais comment les relier ?

Une solution pourrait être de générer un identifiant dans les tables qui n'en ont pas (fig. 4.13). Cependant, cette manœuvre doit se faire à un moment plus ou moins précis de la transformation. En imaginant que l'on ait une règle permettant de créer des identifiants techniques, il faut s'assurer que celle-ci se déclenche après la règle transformant les attributs "unique" en clés primaires. Si ce n'est pas le cas, on pourrait créer, par erreur, des identifiants techniques là où ce n'est pas nécessaire. De même, si l'on considère qu'une sous-classe hérite de l'identifiant de sa super-classe, il ne faut pas générer de clé primaire dans les tables associées aux sous-classes. Cependant, si la table associée à la super-classe n'a pas d'identifiant, il faut lui en créer un d'abord avant de la propager dans les tables associées aux sous-classes. Ce problème complexe des identifiants met en avant un besoin évident de contrôle au niveau du déclenchement des règles.

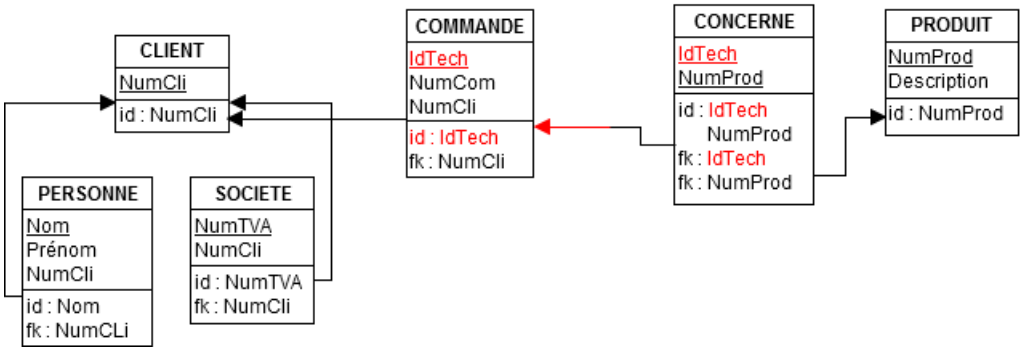

Figure 4.13: *Exemple de schéma relationnel avec des identifiants techniques générés*

Il faut contrôler une séquence de transformations constituées des étapes suivantes:

1. Transformation des attributs "unique" en clés primaires.
2. Créer des identifiants pour les tables qui n'en ont pas et qui ne sont pas des sous-classes.
3. Traiter les relations d'héritage.

Cependant, le fait d'avoir des mécanismes de contrôle ne résout pas tous les problèmes. En effet, si l'on souhaite traiter le problème des identifiants absents, la deuxième étape ci-dessus consiste à vérifier s'il existe des tables qui n'ont pas d'identifiant. Or, la table est un concept associé au méta-modèle cible. Cela implique donc de pouvoir tester des concepts du méta-modèle cible dans les conditions des règles que nous dérivons.

Il est intéressant de remarquer que dans cet exemple des identifiants absents, il est question de repérer les tables qui "n'ont pas" d'identifiant. En d'autres mots, il faut vérifier la non-existence d'un fait relatif à des clés primaires. La solution initiale ne permet pas de générer des conditions testant la non-existence de concepts.

Aussi, on peut remarquer que la deuxième étape de cet exemple vérifie que les tables n'ont pas d'identifiant mais également qu'elles ne sont pas des sous-classes. Cette règle de transformation comporte une condition regroupant plusieurs concepts issus des méta-modèles source et cible. En l'occurrence, il est question du concept "table-colonne" du méta-modèle cible et du concept "héritage" du méta-modèle source. La solution initiale n'est pas capable d'associer plusieurs concepts dans les conditions des règles qu'elle dérive. Cela implique donc de ne pas pouvoir générer des règles de

transformation puissantes.

Enfin, lors d'une transformation de modèles, il peut être intéressant de pouvoir tester certaines propriétés des concepts d'un méta-modèle. Un problème qui nous intéressera est celui des associations de type un-à-un. Bien que le méta-modèle utilisé pour représenter les schémas relationnels permette d'avoir un lien un-à-un entre deux tables, une bonne pratique recommande cependant de "fusionner" ces deux tables en une seule (fig. 4.14) [39]. Cette manipulation préserve la sémantique mais peut-être vue comme une opération de refactoring, puisque, d'une certaine façon, on améliore le modèle en sortie. Afin d'y arriver, il est nécessaire de connaitre le nombre d'associations dans lequel une classe est impliquée. On ne peut en effet "fusionner" une table avec une autre que si elle n'a qu'un et un seul lien de type un-à-un. Mais on peut imaginer avoir besoin d'autres informations telles que le nombre d'attributs d'une classe, le nombre de classes dans un modèles, etc. Ces propriétés sont obtenues grâce à des requêtes que nous appellerons "primitives de navigation". Ces requêtes sont greffées aux conditions des règles, d'où la nécessité de pouvoir construire des règles plus complexes que celles de la solution initiale.

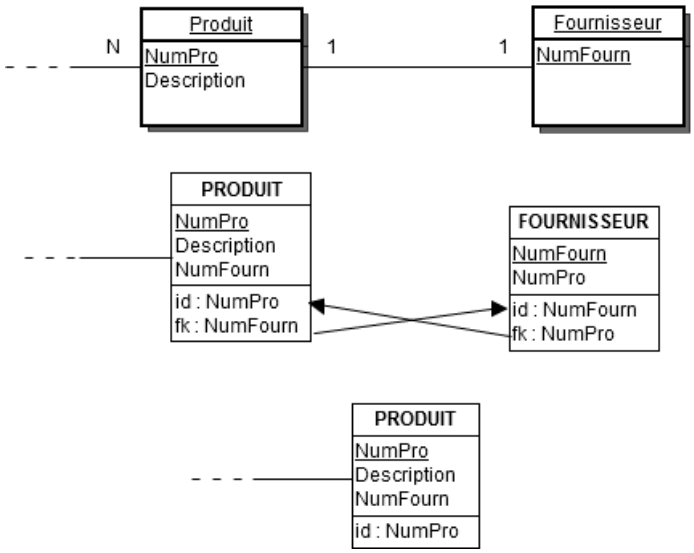

Figure 4.14: *Gestion des associations 1-1*

Note: Pour cet exemple, il est légitime de s'interroger sur l'utilité de tester l'existence d'une clé primaire dans une table, au lieu de tester la présence d'un attribut unique dans la classe associée à cette table, ce qui permettrait dès lors de ne pas tester des concepts du modèle cible. Si l'on procédait de la sorte, des identifiants techniques seraient peut-être ajoutés

aux tables associées à des sous-classes. Quoi qu'il en soit, ce n'est pas le seul exemple qui nécessite de tester des concepts du méta-modèle cible. Cette manipulation nous sera utile pour bien d'autres règles.

Maintenant que les principales limitations de la solution initiale ont été mises en évidence, cinq nouveaux mécanismes vont permettre de les résoudre:

- l'utilisation du contrôle explicite (ou externe) afin de résoudre certains problèmes demandant un plan de transformation ordonné;
- la possibilité de créer des règles de transformation complexes;
- la possibilité de tester le modèle cible dans les conditions des règles;
- la possibilité de tester des propriétés des modèles et constructions;
- la possibilité de vérifier la non-existence d'un fait.

4.5 Évolution de la solution initiale selon deux pistes

Dans cette section, nous présenterons les différentes améliorations qui ont été apportées à la solution initiale dans le but de résoudre les limites identifiées dans la section précédente. Deux approches ont été imaginées. La première approche, que nous appellerons RAC (Rules And Control), fonctionne de façon similaire à la solution initiale. Des améliorations ont été apportées afin de pallier les limitations décrites précédemment. La deuxième approche, appelée RTC (Rules Then Control), fonctionne en deux phases distinctes. La première phase consiste à générer des ensembles de règles et la seconde phase consiste à trouver un ordonnancement optimal dans un ensemble de règles. Bien entendu, les modifications apportées à la méthode RAC seront conservées dans la méthode RTC étant donné que les limitations sont liées à l'approche initiale et que les deux nouvelles approches reposent entièrement dessus.

4.5.1 Approche RAC

Modification des méta-modèles

Alternname Lorsque le modèle source décrit une classe A possédant une et une seule association qui est de type 1-1 vers une autre classe B, la littérature suggère de fusionner la classe A dans la classe B. Ce qui nous donnerait, après la transformation de modèle, une table B dans le schéma relationnel résultant, qui posséderait tous les attributs de la classe B ainsi que tous

les attributs de la classe A. Pour ce genre de transformation, il faut pouvoir générer une règle complexe permettant de reconnaître une classe se trouvant exactement dans cette situation et permettant de réaliser la transformation qui consiste à créer une table pour deux classes et à y transférer tous les attributs. Cette règle est très complexe et malheureusement très difficile à faire apprendre par un programme de façon automatique. Une solution possible consiste en la division de la règle de transformation en plusieurs règles simples. Ces règles devront être ordonnées et utilisées de façon incrémentale durant l'exécution. Nous allons utiliser la traçabilité pour permettre de diviser ce problème.

La traçabilité fournit une trace d'exécution d'une transformation. Une fonctionnalité de "tracing" peut être implémentée directement comme une partie d'une règle de transformation. Une trace peut être stockée dans le modèle source, dans le modèle cible ou dans un endroit séparé. Elle peut être analysée comme un élément du modèle et peut être utilisée comme un élément essentiel dans l'exécution d'une transformation de modèle.

Par rapport aux précédents exemples, nous avons rajouté une nouvelle propriété se nommant "Altername" au constructeur des tables dans le métamodèle du schéma relationnel. Cette propriété va nous permettre de savoir si une table est bien le résultat de la fusion de deux classes.

Lors de l'exécution, cette propriété est utile pour pouvoir transférer tous les attributs de la classe A dans la table B (table résultante de la fusion de la classe A avec la classe B), comme le montre cette règle de transformation qui transfère les attributs d'une classe qui a été fusionnée dans la table résultante de cette fusion:

```
(defrule Transfo−Merged−Attribute−Column
 (attribute (name ?a00)(class ?c00)(unico ?a02))
 (class(name ?c00))

 (table (name ?t20)(altername ?c00))

 (attribute (name ?a40)(class ?t20)(unico 1))
 (class(name ?t20))
 =>
 (assert (column (name ?a00)(table ?t20)(pk 0)))
)
```

Mais elle peut également être utile comme information complémentaire dans le schéma relationnel pour savoir s'il y a eu des fusions de classes lors de la transformation de modèles.

D'une certaine manière, nous avons pu constater qu'il peut être plus facile d'apprendre plusieurs petites règles plutôt qu'une seule mais trop complexe.

Autres modifications Pour diminuer l'espace de recherche, les méta-modèles utilisés dans l'approche précédente ont été modifiés en supprimant les constructeurs des "Foreign Keys" dans le schéma relationnel et des "Uniques" dans le diagramme de classe afin de les rajouter respectivement comme propriétés d'une colonne ou d'un attribut.

Nous avons dû faire un nouveau choix, en rajoutant une nouvelle propriété dans les constructeurs des Associations N-M: le nom du rôle de cette association. Il est nécessaire pour permettre de générer une table d'association lorsqu'il n'existe pas de classe d'association dans le diagramme de classe. Malheureusement, cela augmente la taille de l'espace de recherche.

Voici les méta-modèles finaux qui seront utilisés par la suite:

Diagramme de classe

```
class (name)
attribute (name) (class) (unico)
association1n (classfr) (classto)
associationnm (name) (classfr) (classto) (classas)
association11 (classa) (classb)
inheritance (class) (superclass)
```

Schéma relationnel

```
table (name) (altername)
column (name) (table) (pk)
fk (column) (table) (fktable)
```

Modification du typage des propriétés

Le type *NIL* Dans la version précédente, seuls les types *Integer* et *String* étaient utilisés dans les règles. Malheureusement, ce n'est pas suffisant pour décrire certaines particularités que l'on peut retrouver dans un modèle. Par exemple, dans le méta-modèle du diagramme de classe de l'approche initiale, une association N-M comprend une classe source, une classe cible et une classe d'association. Ce méta-modèle n'est pas complet, car il est possible qu'une association N-M dans un diagramme de classe ne possède pas de classe d'association.

Ce problème peut être résolu de deux manières différentes. Soit on complète le méta-modèle en rajoutant un constructeur "*AssociationNMWithoutClassAs*" décrivant une association N-M qui ne possède pas de classe d'association, soit on rajoute un type qui aurait comme ensemble de valeur, toutes les valeurs du type *String* mais également une constante spécifique qui indique l'absence de valeur. Ce type sera appelé *NIL*.

La première solution a été écartée car l'ajout d'un nouveau constructeur augmente l'espace de recherche de façon beaucoup plus importante que l'ajout d'une constante pour certaines propriétés. Il est également à constater que si un élément d'un modèle possède N attributs facultatifs, il faudrait ajouter au méta-modèle toutes les possibilités, c'est-à-dire, 2^N constructeurs. Le type *NIL* a donc été retenu pour pouvoir représenter des propriétés qui peuvent ne pas posséder de valeur.

Les types et les domaines Dans un méta-modèle, on définit, pour chaque propriété d'une entité et pour chaque garde, les types de valeurs qu'elles peuvent recevoir. La nouvelle approche compte quatre types de valeurs : *Integer, String, ATOM, NIL.*

ATOM est un type défini par JESS qui correspond au type *Boolean* en Java. Il peut posséder deux valeurs: 0 et 1. Il est utilisé par exemple pour indiquer si un attribut est Unique ou si une colonne est une Primary Key.

Le type *NIL* correspond à l'union des valeurs du type *String* et de la constante *NIL*. Le type *NIL* étant un ensemble plus grand, la modification de certaines propriétés de type *String* en type *NIL* dans les méta-modèles de l'approche initiale a augmenté de façon significative l'espace de recherche. Mais vu que le type *String* est un sous-ensemble du type *NIL*, il sera possible de jouir de cette relation pour permettre de lier des variables de type *String* avec une variable de type *NIL*.

On définit également un domaine pour chaque entité du modèle. Le méta-modèle du schéma relationnel possède par exemple trois constructeurs (entités) : *table, column, fk.*

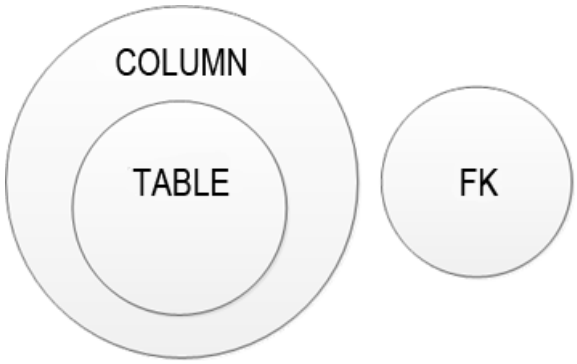

Figure 4.15: *Les domaines du schéma relationnel*

Le schéma ci-dessus (fig. 4.15) indique qu'on peut insérer une valeur de table dans une colonne. Par contre, l'inverse n'est pas permis.

Depuis que nos règles de transformation permettent de tester un fragment du modèle cible dans leur partie gauche, l'utilisation des domaines a été utilisée afin de réduire l'espace de recherche et d'éviter d'avoir des tables ayant des noms de colonnes. Par contre, il est permis de générer des règles pouvant faire l'inverse (nom de table comme nom de colonne). Cela à cause du choix de prendre le nom des tables pour générer les identifiants techniques.

Changement dynamique de types Reprenons l'exemple de l'association N-M qui possède une propriété *classas* de type *NIL*. Pour les associations N-M qui possèdent une classe d'association, le programme devra générer une règle qui transforme l'association en une table ayant le nom de la classe d'association. Pour les associations N-M qui ne possèdent pas de classe d'association, le programme devra générer une règle qui transforme l'association en une table ayant le nom du rôle de l'association.

Deux problèmes ont été rencontrés pour ce type de règle. Premièrement, on constate que le fait qu'une variable associée à une propriété de type *NIL* (*classas*) possède ou pas de valeur influence le comportement de la transformation. Mais comment savoir si cette variable possède ou pas une valeur ? Il faut ajouter la possibilité de faire des tests sur les variables associées aux propriétés de type *NIL*. JESS fournit un mécanisme permettant d'y arriver:

```
(test (neq ?s03 NIL))
```

Cette expression teste le fait que la variable s03 n'est pas équivalente à *NIL*, et donc qu'elle contient une valeur.

Deuxièmement, on constate qu'une table peut posséder le nom d'une classe d'association. Malheureusement le nom d'une table dans le méta-modèle du schéma relationnel est de type *String*. Ce qui est logique car une table doit toujours posséder un nom. Pour résoudre ce dernier problème, le test vérifie si une variable possède une valeur. Si celui-ci est positif, le type (préalablement *NIL*) va être modifié pour permettre à cette dernière d'être utilisée dans la partie droite de la règle comme une variable de type *String*.

Le contrôle

Les langages de transformation utilisent différents mécanismes pour savoir quand une règle de transformation est appliquée:

Le contrôle implicite: permet seulement de contrôler l'application des règles lors de l'exécution, par contre, il ne permet pas de spécifier l'ordre d'application des règles de transformation.

Le contrôle explicite: permet de définir les règles à appliquer et en même temps de spécifier l'ordre d'exécution des règles.

Le contrôle externe: va seulement spécifier l'ordre d'exécution après que les règles soient sélectionnées.

JESS fournit deux mécanismes de contrôle explicite:

Les modules: les modules consistent en des sous-ensembles de règles du programme de transformation. Les modules portent des noms qui permettent de les identifier. Les règles se trouvant dans un module peuvent être exécutées/déclenchées uniquement si le module dans lequel elles se trouvent dispose du focus. Le focus ne peut être détenu que par un seul module à la fois. Les modules sont placés dans une pile et le module situé en haut dispose du focus. Une fois que les règles du module situé en haut de la pile ne peuvent plus être déclenchées, le module est retiré de la pile et celui en dessous hérite du focus. Cette pile évolue de façon automatique mais peut être manipulée avec des commandes JESS.

La salience: la salience est une propriété des règles qui agit comme un paramètre de priorité. Les règles ayant été activées sont tirées en fonction de leur valeur de salience. Ainsi, celle ayant la plus grande valeur de salience sera exécutée avant les autres.

Modification de l'encodage des règles et des programmes

Les règles disposent également de nouvelles propriétés permettant la prise en charge du contrôle. Ainsi, les règles disposent d'un numéro de module et d'un numéro de priorité. Au fil des générations, ces numéros changeront. Cependant, un seul des deux numéros est utilisé à la fois, bien qu'il soit possible d'utiliser à la fois les priorités et les modules simultanément...

Les conditions des règles

Les briques Les briques sont les blocs de constructions utilisés pour construire la partie conditionnelle des règles. Une règle peut comporter plusieurs conditions afin de pouvoir créer des règles plus complexes que dans la solution initiale.

Les briques sont prédéfinies par l'utilisateur et sont constituées d'un ou plusieurs faits élémentaires. Les briques représentent des concepts du méta-modèle. Par exemple, il est possible de définir une brique permettant de représenter le concept d'héritage entre deux classes (diagramme de classe). Dans le méta-modèle utilisé, un constructeur permet d'exprimer la notion d'héritage:

```
inheritance (class) (superclass)
```

Mais la sémantique du méta-modèle stipule qu'une relation d'héritage est établie entre une classe et une super-classe. L'élément inheritance n'est pas suffisant à lui-même car il dépend de deux classes. Dès lors, pour définir le concept d'héritage, il est nécessaire d'ajouter les deux classes, ce qui mène à la brique suivante:

```
(inheritance (class ?c20)(superclass ?c10)))
(class(name ?c10))
(class(name ?c20))
```

Il en va de même avec le concept d'attribut appartenant à une classe. Un attribut ne peut exister sans la classe à laquelle il appartient. La brique suivante définit le concept d'attribut:

```
(attribute (name ?a00)(class ?c00)(unico ?a02))
(class(name ?c00))
```

C'est pareil pour le concept d'association:

```
(association11 (classa ?c00)(classb ?c10))
(class(name ?c00))
(class(name ?c10))
```

Il arrive parfois qu'une condition porte simplement sur une classe. Une brique est donc créée pour exprimer le concept de classe, bien que celle-ci ne contienne qu'un seul élément: la classe.

Comme on peut le voir, un élément/fait est commun à toutes les briques définies pour exprimer des conditions sur les diagrammes de classe. Il s'agit de la classe. Ce constructeur est différent des autres car il est auto suffisant, c'est-à-dire qu'il ne dépend d'aucune autre construction, contrairement aux attributs, à l'héritage ou aux associations.

Dans le schéma relationnel, on définit également des briques pour exprimer les concepts du schéma relationnel, à savoir: les tables, les colonnes, et les clés étrangères. Tout comme les classes, les tables sont auto suffisantes.

Afin de garder une cohérence interne dans les briques, il faut établir des liens entre les faits qui les composent. On parle alors de liens internes. Les constructions n'étant pas auto suffisantes sont reliées aux constructions qui le sont. Ainsi, dans la brique de l'héritage, inheritance est relié à deux classes. Il en va de même pour les associations et l'attribut. La liaison d'un constructeur (esclave) vers un autre constructeur (maître) consiste à remplacer le nom d'une des variables de l'esclave par le nom d'une des variables du constructeur maître. Bien entendu, les variables doivent être compatibles en domaine et en type.

Les conditions complexes La structure utilisée pour les règles de transformation permet d'éviter de créer des fragments de modèle syntaxiquement incorrects. Malheureusement, ces règles ne sont pas assez raffinées pour pouvoir générer un plan de transformation idéal. Il a été constaté qu'un élément du modèle source ne correspond pas toujours à un élément spécifique du modèle cible. Parfois, un élément du modèle source pourra être transformé en un élément du modèle cible selon le contexte dans lequel il se trouve.

Nous devons donc redéfinir la structure utilisée pour générer des règles de transformation. Une règle doit pouvoir non seulement vérifier si un fait existe mais aussi pouvoir vérifier si le fait se trouve dans le contexte voulu. Une règle de transformation peut maintenant générer des conditions plus complexes. C'est-à-dire que sa partie gauche peut posséder plusieurs "briques",

comme le montre la règle suivante, où celle-ci possède une brique attribut et une brique table:

```
(defrule Transfo−Attribute−2−Column
 (attribute (name ?a00)(class ?c00)(unico 0))
 (class(name ?c00))

 (table (name ?c00)(altername ?t21))
 =>
 (assert (column (name ?a00)(table ?c00)(pk 0)))
 (assert (table (name ?c00)(altername nil))
)
```

Les conditions sur le méta-modèle cible Une règle de transformation de modèle décrit comment un fragment du modèle source peut être transformé en un fragment du modèle cible. Généralement, une règle contient deux parties. La première décrit le fragment du modèle source et est appelée la partie gauche de la règle. La seconde partie décrit le fragment à créer dans le modèle cible lors de l'exécution, c'est la partie droite. Dans notre contexte, cette description d'une règle de transformation n'était pas suffisante. Dans certain cas, malgré un bon ordonnancement de l'exécution des règles, certaines règles ont besoin de vérifier si certains fragments ont déjà été créés dans le modèle cible. Par exemple, le fait de vérifier s'il existe des tables qui ne possèdent toujours pas d'identifiants après avoir transformé les Unique en Primary Key. Pour cela, nous avons étendu la partie "condition" des règles afin qu'elles puissent accepter la description de fragment du modèle cible. Ce type de règles vérifie l'existence d'un fragment du modèle source tout en vérifiant si une partie du modèle cible existe déjà afin de compléter correctement le modèle cible avec un nouveau fragment. Ces règles ne seront jamais exécutées au début de l'exécution à cause de la condition portant sur le modèle cible. On peut donc voir que ce type de règles peut également être utilisé pour réaliser un semblant d'ordre dans l'exécution.

L'espace de recherche a été fortement augmenté du fait de la possibilité de faire des tests sur des fragments du modèle cible. Mais, d'un autre côté, cela permet de construire des règles moins complexes et plus faciles à apprendre car la vérification portant sur un fait déjà asserté permet d'éviter de retester toutes les conditions qui ont été établies pour l'asserter.

Prenons comme exemple la génération des identifiants techniques. Ces identifiants peuvent être générés sur le modèle cible seulement s'il existe une table qui ne possède pas de clé primaire et dont la classe correspondante dans

le modèle source ne possède pas de super-classe.

```
(defrule Generate−PK
 (table (name ?t00)(alternname ?t01))

 (NOT (column (name ?o20)(table ?t00)(pk 1)))
 (table (name ?t00)(alternname ?t21))

 (NOT (inheritance (class ?t00)(superclass ?c50)))
 (class(name ?t00))
 =>
 (assert (column (name ?t00)(table ?t00)(pk 1)))
)
```

Dans cette règle, la partie gauche est divisée en trois conditions. Deux concernant le modèle cible et une concernant le modèle source. A cette étape du plan de transformation, les tables ont déjà été créées dans le modèle cible. La première condition permet de vérifier si la table *t00* existe et n'a plus besoin de vérifier si une telle table peut exister selon le modèle source. La deuxième condition permet de savoir si toutes les étapes qui ont précédé dans le plan de transformation ont permis ou non de créer une clé primaire dans la table *t00*. La dernière condition est une "brique" traditionnelle qui utilise le méta-modèle source. Elle vérifie si la classe *t00* ne possède pas de super-classe.

Cette règle révèle plusieurs points importants:

Premièrement, la possibilité de pouvoir tester une partie du modèle cible permet d'éviter de nombreuses revérifications et allège donc les conditions nécessaires. Plus une règle est complexe, plus la probabilité que notre programme l'apprenne est faible. L'espace de recherche a pu être réduit grâce à cela. Les règles de transformation élaborées dans le plan de transformation idéal, c'est-à-dire, le programme idéal que l'on souhaite générer pour la transformation d'un modèle de type diagramme de classe en un modèle de type schéma relationnel, ne possèdent au maximum que trois conditions dans leur partie gauche (voir chapitre suivant).

Deuxièmement, on constate que cette règle s'exécute avec l'hypothèse que certaines transformations se sont déjà exécutées. Pour pouvoir émettre de telles hypothèses, il est impératif que les règles de transformation suivent un ordre bien défini et donc un contrôle explicite est manifestement nécessaire.

Finalement, nous pouvons émettre l'hypothèse que notre approche peut également servir pour des transformations de type endogène. Ce système pourrait donc être utilisé pour d'autres types d'activités notamment le refac-

toring, la normalisation et le raffinement de modèle.

Liaisons des conditions (inter briques) Pour vérifier dans quel contexte se trouve un fait, les différentes "briques" utilisées dans les règles de transformation doivent être reliées entre elles. Il est clair qu'une règle de transformation n'a pas de sens si les conditions de déclenchement n'ont aucun rapport entre elles. Elle doit vérifier l'existence sur le modèle d'un fragment uniforme qu'il soit petit ou grand. Il est donc nécessaire d'imposer une liaison entre chaque "briques" afin de ne pas vérifier l'existence de deux fragments qui n'ont aucun lien sur le modèle.

Comme il a été constaté, les règles de transformation générées peuvent être composées de deux types de "briques". Les briques qui vérifient l'existence d'un fait dans le modèle source et les briques qui vérifient si un fait a déjà été inséré dans le modèle cible.

A cause des conditions portant sur le modèle cible dans la partie gauche de la règle, deux types de liaisons ont dû être implémentées:

- Les liaisons qui relient deux briques portant sur le même méta-modèle (donnant des liens de type $SP - SP$ ou $TP - TP$).

- Les liaisons qui relient deux briques, l'une portant sur le méta-modèle source et l'autre portant sur le méta-modèle cible (donnant des liens de type $SP - TP$ ou $TP - SP$).

Les briques utilisant le même méta-modèle peuvent être reliées à l'aide d'un constructeur commun. Ce constructeur ne doit pas être pris au hasard car il doit assurer l'existence de cette liaison entre les deux fragments décrits par les briques.

Par exemple, prenons le constructeur définissant un "attribut" comme lien entre deux briques décrivant deux faits appartenant à un diagramme de classe. Cet attribut doit être commun aux deux briques. On pourrait simplement relier les briques à l'aide du nom de l'attribut, mais malheureusement, il est possible que deux attributs différents dans un diagramme de classe possèdent le même nom car deux classes différentes peuvent posséder chacune un attribut nommé de manière identique. Ces attributs n'ont généralement rien en commun et sont indépendants l'un de l'autre. Deux briques ne peuvent donc être reliées à l'aide du nom d'un attribut. Pour assurer qu'une liaison soit pertinente, le lien ne doit pas être ambigu. Pour qu'un attribut serve de liaison, il faut que dans les deux briques, les attributs possèdent le même nom et la même classe. Pour cela, il faut que les deux briques soient identiques et il n'est évidemment pas très utile de vérifier

deux fois le même fait.

Pour relier deux faits différents ayant un lien existant réellement dans le modèle, il faut les relier avec un constructeur ou un slot qui peut être identifié de façon non ambiguë sur le modèle. Les seuls constructeurs ayant la propriété de décrire complètement un fait du modèle sans l'aide d'autres constructeurs sont les constructeurs auto-suffisants. Si un constructeur peut à lui seul créer une brique, il est auto-suffisant. De la même manière, les slots pouvant à eux seuls identifier une entité sur le modèle de façon non ambiguë sont des slots auto-suffisants.

Pour les diagrammes de classe, seul le constructeur *class* et le slot *classas* appartenant au constructeur *associationnm* peuvent servir de lien. Pour les schémas relationnels, ce sont les constructeurs de type *table*.

Dans l'exemple suivant, le slot *classas* du constructeur*associationnm* est relié au slot *name* du constructeur *class* de la deuxième brique.

```
(defrule Transfo−ClassAs−Attribute
 (associationnm (name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
 (class(name ?c00))
 (class(name ?c10))

 (attribute (name ?a20)(class ?s03)(unico ?a22))
 (class(name ?s03))
 =>
 (assert (column (name ?a20) (table ?s03)(pk ?a22)))
)
```

En revanche, il est parfois nécessaire de relier deux briques de méta-modèles différents, lorsque l'une des conditions de la règle porte sur le méta-modèle cible. Dans ce cas-là, il n'est pas possible de relier les briques via des constructeurs communs, puisque les méta-modèles sont différents. Il faut les relier via une propriété qui est de même type.

Algorithme de liaison du coté gauche La création d'une règle, ou sa modification durant le processus d'évolution consiste à assembler des briques de conditions (aléatoires) dans le côté gauche et des faits (aléatoires également) dans le côté droit.

Comme expliqué précédemment, les liaisons entre briques se font via des constructions communes s'il s'agit de relier deux briques de même méta-modèle ou via des propriétés de types compatibles s'il s'agit de briques de méta-modèles différents.

Cependant, les liaisons entre ces briques ne peuvent pas se faire n'importe comment. De plus, le processus de liaison peut se produire dans trois cas: la création d'une règle, l'ajout d'une condition et la suppression d'une condition. Cependant, la création d'une règle peut être vue comme l'ajout successif de conditions. Il est possible que, lors d'une suppression de condition, aucune liaison ne soit nécessaire. En effet, si on représente les conditions sous forme d'arbre, on peut voir que la suppression des "conditions-feuilles" est sans conséquence pour la règle. En revanche, si on supprime une racine, il sera nécessaire de "recoller" les conditions filles (fig. 4.16).

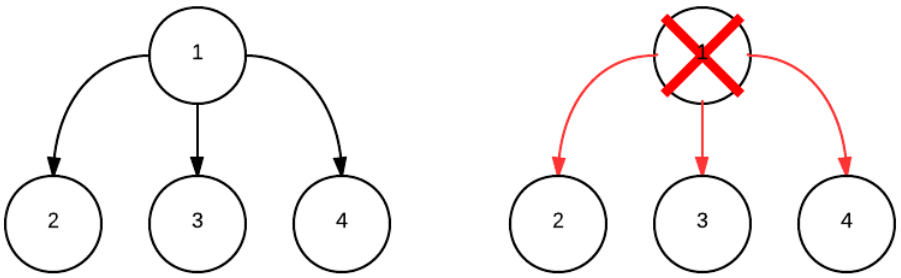

Figure 4.16: *Graphe des liaisons: suppression d'une condition racine*

De plus, une fois qu'une liaison est établie, la brique "maitre" propage la variable servant de liaison à la brique "esclave". Il faut donc veiller à éviter des cycles dans les conditions.

La représentation en arbre permet de se rendre compte que s'il y a N conditions dans la règle, alors il y aura forcément N-1 liaisons entre les briques.

L'algorithme de liaison des briques lors de la création d'une règle se présente alors comme suit:

```
NbArc := NbBriques -1;
choisir une brique “maître” M dans { Briques }
choisir une brique “esclave” E dans { Briques } \ M
relier M à E
ajouter M et E dans { Briques liées }
NbArc := NbArc -1
Tant que NbArc > 0
        Choisir une brique “maître” M dans { Briques liées }
        Choisir une brique “esclave” E dans { Briques } \ { Briques liées}
        relier M à E
        ajouter E dans { Briques liées }
        NbArc := NbArc -1
```

Le fait de choisir une brique “maître” dans l'ensemble Briques liées est nécessaire car il se pourrait que l'on sélectionne une brique dans l'ensemble de départ et qu'on le relie à l'esclave de l'itération précédente, modifiant alors les liaisons ayant été effectuées à l'itération précédente. Sur le schéma suivant (fig. 4.17), l'ajout de la brique n°3 en tant que maître risquerait d'altérer les liaisons ayant été éffectuées entre les briques 1 et 2.

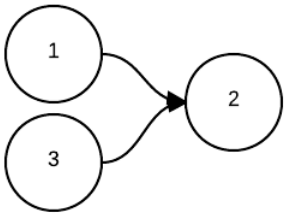

Figure 4.17: *Graphe des liaisons: deux maîtres pour un seul esclave*

Le sens des liaisons est laissé au hasard mais est tout de même contrôlé de façon à éviter ce genre de situation problématique.

Lors de la suppression d'une brique ayant plus d'un arc, il se peut qu'il faille relier une brique à un ensemble de briques déjà reliées, comme le montre la figure suivante (fig. 4.18).

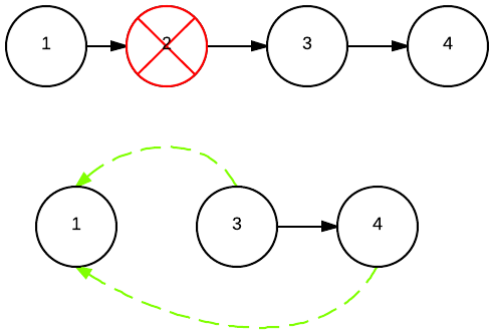

Figure 4.18: *Graphe des liaisons: suppression d'une brique ayant plus d'un arc*

Mais on voit très bien que l'on retombe dans le même cas que pour la création d'une règle, où il suffit de relier la brique isolée à un ensemble de briques déjà liées au moyen d'un seul arc. Il est important de noter qu'il n'y a pas de logique dans la liaison en cas de suppression. Dans l'exemple précédent, il pourrait sembler logique de relier la brique 1 à la brique 3, cependant, cela implique de devoir propager les variables de 1 vers 3, puis de repropager cette modification vers 4, au cas où il y aurait eu un changement de la variable de liaison entre 3 et 4.

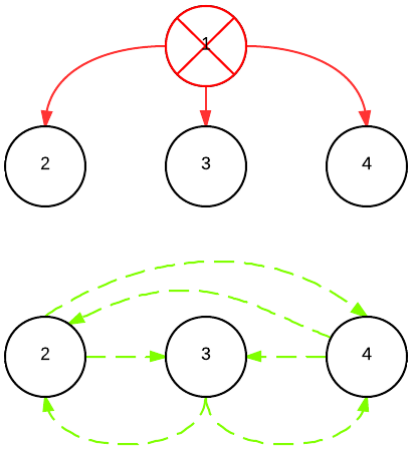

Figure 4.19: *Graphe des liaisons: sens de liaison possibles*

On peut voir sur le schéma ci-dessus (fig. 4.19) que les liaisons peuvent se faire dans n'importe quel sens (au début de la suppression).

Algorithme de liaison du coté droit

Les actions des règles

Les faits Une règle de transformation produit des faits afin de compléter le modèle résultant du programme de transformation. Il est à constater que dans une règle standard de transformation, les faits produits correspondent à des blocs unis du schéma attendu. Pour cette raison, les règles générées par notre programme posséderont des faits qui seront également reliés entre-eux. Pour relier deux faits, ils doivent posséder au moins une propriété partageant le même domaine et le même type de valeur. La liaison peut se faire entre un nombre quelconque de propriétés compatibles. Ce nombre est défini aléatoirement et les bornes vont donc de un aux nombres de propriétés compatibles entre les deux faits.

Calcul des variables Lors de la création des règles, la partie condition est créée au hasard, tout comme la partie action. Cependant, cette dernière dépend de la partie condition. En effet, dans la partie condition, les variables sont créées selon les besoins des constructeurs. Les actions étant une réponse aux conditions, celles-ci doivent utiliser des variables issues de la partie condition.

```
(defrule Gen−PK
 (table (name ?t00)(alternname ?t01))

 (not (column (name ?o20)(table ?t00)(pk 1)))
 (table (name ?t00)(alternname ?t21))

 (not (inheritance (class ?t00)(superclass ?c50)))
 (class(name ?t00))
 =>
 (assert (column (name ?t00)(table ?t00)(pk 1)))
)
```

Dans la règle ci-dessus, plusieurs variables sont déclarées. Elles constituent un premier "ensemble" de terminaux (fig. 4.20).

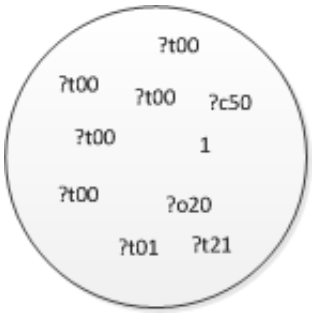

Figure 4.20: *Ensemble de terminaux non normalisé*

Ce n'est pas réellement un ensemble puisqu'il existe des doublons. De plus, il n'est pas totalement complet, puisque certaines constructions du méta-modèle cible peuvent prendre les valeurs entière 0 et 1 (les colonnes) ou des valeurs *NIL* (les tables). Il est donc nécessaire de "normaliser" cet ensemble. Il est nécessaire de normaliser cet ensemble (fig. 4.21).

Figure 4.21: *Ensemble de terminaux normalisé*

Une fois cet ensemble créé, l'algorithme choisit au hasard des variables compatibles avec les propriétés des constructions du méta-modèle cible. On ne peut, en effet, pas mettre un 0 comme nom d'une table (type checking). La normalisation de l'ensemble, outre l'ajout des variables, se charge d'éliminer les doublons. En effet, elle est nécessaire car sans elle, certaines variables auraient plus de chance d'être sélectionnées que d'autres, rendant la recherche dans l'espace de solution plus difficile.

Les variables sont des objets, et même si elles ont le même nom, elles peuvent différer au niveau de leurs autres propriétés et certaines de ces propriétés prévalent sur d'autres. Certaines variables sont tirées de constructions issues du modèle cible. Celles-ci auront priorité sur les variables du même nom, mais issues du modèle source. Aussi, à nom égal, on prend la variable dont le domaine est le moins restrictif. La variable *c00* récupérée

dans le nom d'une table peut servir de nom de table OU de nom de colonne. Mais une variable *c00* récupérée dans le nom d'une colonne ne peut être utilisée que dans une autre colonne. C'est donc la variable *c00* issue de la table qui sera retenue. Ces deux critères de choix permettent de laisser plus de possibilité de liaisons entre le coté condition et le côté action.

Test de la non-existence d'un fait

Malgré le fait que les règles de transformation puissent maintenant faire des vérifications un peu plus poussées sur les modèles en utilisant plusieurs conditions dans la partie gauche, il a été constaté que ce n'était pas suffisant pour assurer certaines transformations.

Si par exemple, on souhaite transformer deux classes *c00* et *c10* en une seule table de nom *c10* si ces deux classes sont reliées par une association de type 1-1 et que la classe *c00* ne possède aucune sous-classe, alors il est nécessaire de pouvoir vérifier la non-existence d'un fait. Ici, il faut s'assurer que le modèle ne possède pas de classe ayant pour super-classe la classe *c00*.

Pour tester la non-existence d'un fait, un nouvel opérateur a été implémenté, l'opérateur *NOT*. Lorsqu'une brique doit vérifier si un fait n'existe pas, l'opérateur *NOT* est placé sur le premier constructeur de celle-ci.

```
(defrule Detect−Class−Assoc−11
 (association11 (classa ?c00)(classb ?c10 ))
 (class(name ?c00))
 (class(name ?c10))

 (NOT (inheritance (class ?c20)(superclass ?c00)))
 (class(name ?c00))
 =>
 (assert (table (name ?c10)(alternname ?c00)))
)
```

Le *NOT* est un opérateur permettant de vérifier l'absence d'un fait dans la mémoire de travail de JESS. Cependant, son utilisation peut être coûteuse et une certaine optimisation est nécessaire afin de créer des règles efficaces. En effet, prenons un exemple pour illustrer un cas de *NOT* non-optimisé:

```
(defrule Detect−Class−Assoc−11
 (association11 (classa ?c00)(classb ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (NOT (inheritance (class ?c20)(superclass ?c10)))
 (class(name ?c10))
 (class(name ?c20))
 =>
 (assert (table (name ?c00)(altername ?c10)))
)
```

Dans cette règle visant à détecter les classes impliquées dans des relations 1-1, on veut également vérifier le fait qu'elles ne soient pas impliquées dans des relations d'héritage. Comme il s'agit de conditions, les briques du méta-modèle source sont utilisées pour exprimer ces conditions et lorsque l'on veut en nier une, on se contente d'ajouter un *NOT* sur le constructeur maître de la brique, à savoir inhéritance dans notre exemple. L'exécution de cette règle sur un de nos exemples (fig. 4.22) donne le résultat suivant:

```
?c10 = classe6 ?c20 = classe10
(assert (table (name classe1)(altername classe6)))
?c10 = classe6 ?c20 = classe9
(assert (table (name classe1)(altername classe6)))
?c10 = classe6 ?c20 = classe6
(assert (table (name classe1)(altername classe6)))
?c10 = classe6 ?c20 = classe5
(assert (table (name classe1)(altername classe6)))
?c10 = classe6 ?c20 = classe4
(assert (table (name classe1)(altername classe6)))
?c10 = classe6 ?c20 = classe3
(assert (table (name classe1)(altername classe6)))
?c10 = classe6 ?c20 = classe2
(assert (table (name classe1)(altername classe6)))
?c10 = classe6 ?c20 = classe1
(assert (table (name classe1)(altername classe6)))
```

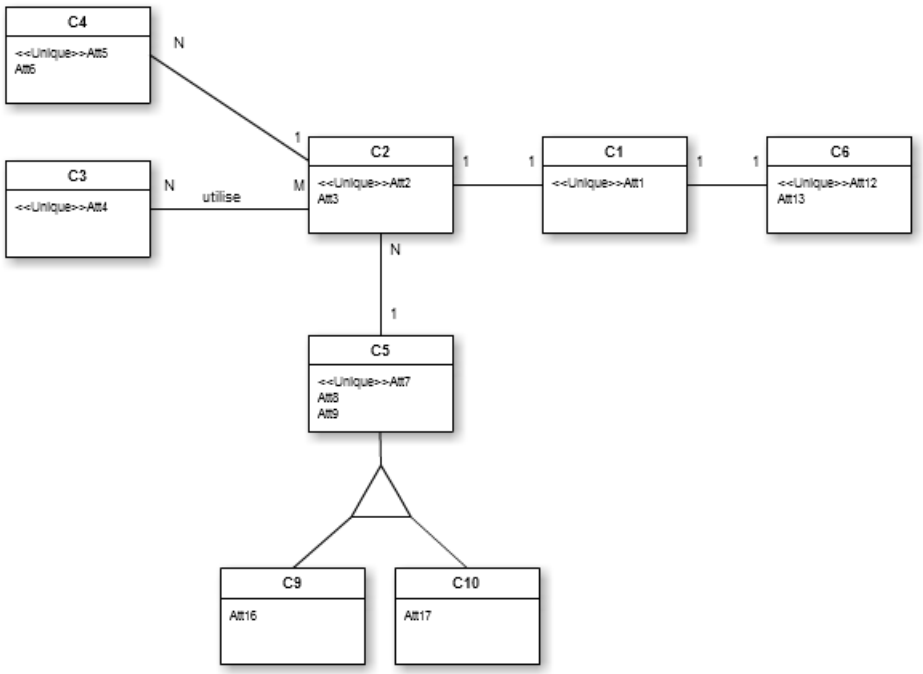

Figure 4.22: *Exemple d'optimisation du NOT*

Le moteur cherche les instanciations des variables qui peuvent déclencher la règle. De ce fait, il trouve huit combinaisons possibles pour les variables *c10* et *c20*. En effet, les classes 1, 2, 3, 4, 5, 6, 9 et 10 satisfont la condition portant sur l'héritage puisque ces classes ne sont en effet pas des sous-classes de la classe 6. Dès lors, le moteur peut activer cette règle huit fois et produire huit fois le même assert.

L'opérateur *NOT* que propose JESS peut ne pas avoir exactement le même comportement qu'un "il n'existe pas" dans la logique des prédicats si ce dernier n'est pas utilisé correctement. Il faut savoir que l'opérateur *NOT* n'instancie pas les variables sur lesquelles il porte et donc n'instanciera pas *c20* ni *c10*.

Les valeurs *c10* et *c20* peuvent par contre être instanciées à l'aide des autres constructeurs. Malheureusement, l'utilisation de la brique décrivant l'héritage utilisée dans l'exemple du cas d'un *NOT* non optimisé instancie bel et bien la variable *c20*.

Pour éviter ce genre de problème, et permettre de générer tous les tests de non-existence possible, la construction des briques utilisant l'opérateur *NOT* a été modifiée. Seuls le constructeur sur lequel l'opérateur *NOT* est

appliqué et les constructeurs servant de liaison avec d'autres briques seront utilisés pour fabriquer la brique. Voici un exemple de la même règle, mais avec un *NOT* optimisé:

```
(defrule Detect−Class−Assoc−11
 (association11 (classa ?c00)(classb ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (NOT (inheritance (class ?c20)(superclass ?c10)))
 (class(name ?c10))
 =>
 (assert (table (name ?c00)(alternname ?c10)))
)
```

Et voici le résultat associé à cette règle:

```
?c10 = classe6
(assert (table (name classe1)(alternname classe6)))
```

Comme on peut le voir, le nombre de possibilités est fortement réduit puisque le moteur ne cherche plus que les classes étant des super-classes.

Cette modification va apporter une nouvelle contrainte à la génération des règles: la partie gauche d'une règle doit posséder au moins une brique qui vérifie l'existence d'un fait (brique sans *NOT*). Effectivement, si toutes les briques vérifient la non-existence d'un fait, aucune variable de la partie gauche de la règle ne sera instanciée et il ne sera pas possible de générer une partie droite pour notre règle.

Primitives de navigations

Les primitives de navigation sont représentées ici sous la forme de requêtes faites auprès du moteur d'inférence de JESS. Ces requêtes permettent, entre autres, de tester le cardinal d'ensembles particuliers. Par exemple, on aimerait pouvoir repérer les classes dans un diagramme de classes qui ne sont reliées qu'à une et une seule autre classe, et que ce lien soit de type 1-1. Cela consiste donc à faire une requête auprès du moteur d'inférence du type: "Quelles sont les classes impliquées dans une association 1-1 ET quelles sont les classes n'ayant qu'une seule association ?". En JESS cela donne:

```
(class (name ?x))
(or (association11 (classa ?x)(classb ?y)) (association11 (classa ?y)(classb ?x)))
(test (eq (count−query−results allAssociations ?x) 1))
```

La primitive de navigation est ici représenté par “count-query-results allAssociations ?x”. *allAssociations* est une defquery, c'est-à-dire la requête que l'on envoie au moteur. Elle est définie comme suit:

```
(defquery allAssociations
 (declare (variables ?c))
  (or (association11 (classb ?c))
  (or (association11 (classa ?c))
  (or (association1n (classfr ?c))
  (or (association1n (classto ?c))
  (or (associationnm (classfr ?c))(associationnm (classto ?c)))))))
)
```

En lui associant une classe *c*, le moteur d'inférence va rassembler tous les faits concordant avec les faits spécifiés dans la requête. L'utilisation de *count-query-result* nous permet de compter le nombre de faits qui ont matchés. Donc, lorsque cette commande nous renvoie 1 comme réponse, on sait que la classe *c* n'a qu'une seule association. De plus, si cette classe *c* est impliquée dans une association de type 1-1 (du coté cible ou source), alors on se retrouve dans une configuration où la classe *c* peut être fusionnée avec la classe à laquelle elle est attachée (fig. 4.23).

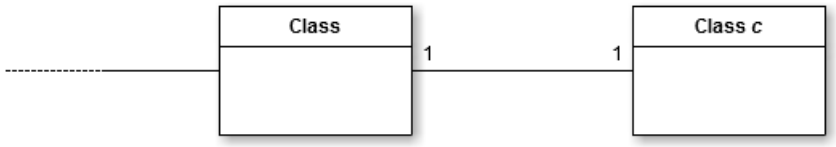

Figure 4.23: *Fusion possible entre deux classes*

Les nouvelles mutations

Après le croisement, les individus ont une probabilité d'être mutés. Les mutations ont pour but d'introduire du nouveau matériel génétique dans les règles, de manière aléatoire. Plusieurs types de mutations ont été définis: Certaines mutations touchent le programme, c'est-à-dire l'ensemble des règles, d'autres touchent les règles en elles-mêmes et certaines mutations sont spécifiques au contrôle des règles.

Mutations des programmes Un programme consiste en un ensemble de règles. Un ensemble est une collection ou un groupement d'objets distincts. Le nombre de règles du programme est le cardinal de cet ensemble et les mutations de programme consistent à faire varier ce cardinal au moyen de deux opérations:

Ajouter une règle: Une règle est créée aléatoirement, de la même façon que durant la phase de création de la population initiale. Cette règle est ensuite ajoutée à l'ensemble des règles et la taille de l'ensemble augmente de un.

Supprimer une règle: Une règle parmi celles figurant dans l'ensemble est sélectionnée aléatoirement et est purement et simplement supprimée. Le cardinal de l'ensemble diminue donc de un. Cependant, cette mutation ne peut être appliquée que si le cardinal est strictement supérieur à un.

Mutations des règles Une règle consiste en un ensemble de briques (dans sa partie gauche - Conditions) et en un ensemble de faits assertés (dans la partie droite de la règle - Actions). Les briques consistent en des ensembles de constructions des méta-modèles. Les briques sont connectées entre elles et les éléments qu'elles contiennent sont également connectés entre eux (fig. 4.24).

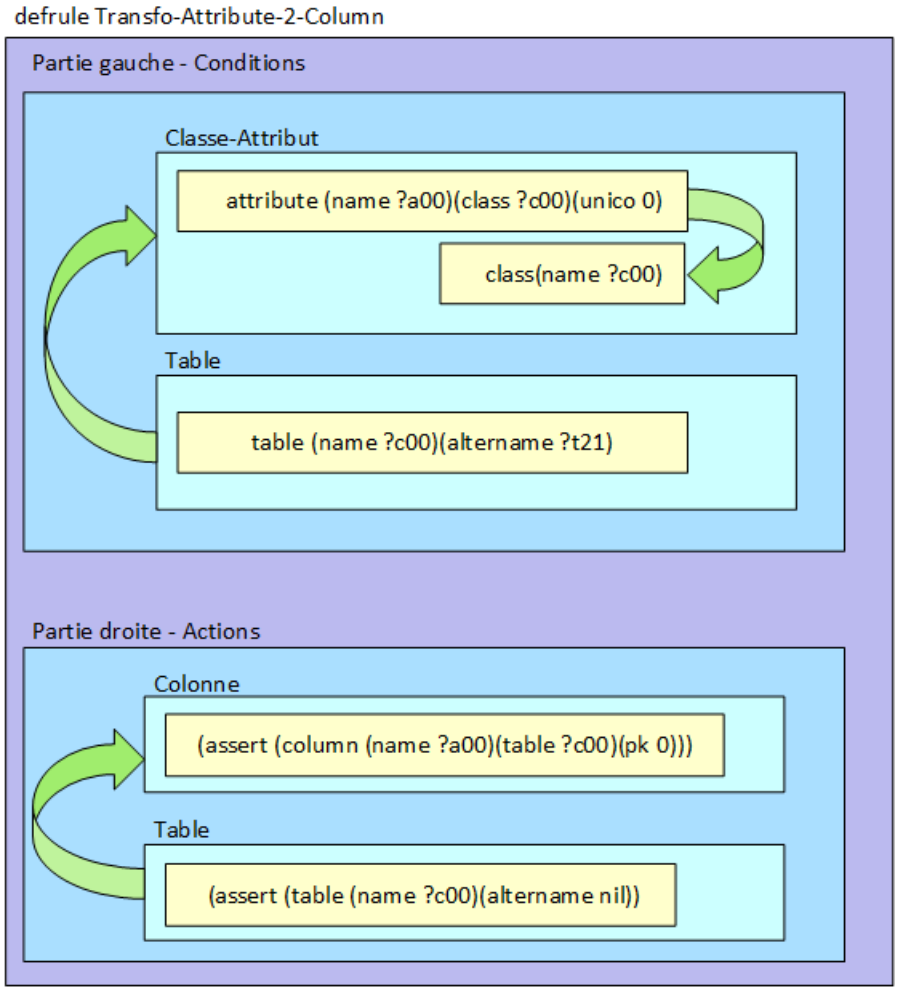

Figure 4.24: *Connexions internes et externes des briques*

Les briques du côté gauche contiennent des variables qui sont utilisées dans le côté droit de la règle. Plusieurs mutations ont donc pu être définies et permettent de modifier les règles à ces différents niveaux:

Ajouter une collection à gauche: Puisque le côté gauche est un ensemble de conditions, il est possible d'imaginer une mutation qui ajoute un élément/une condition à cet ensemble. Bien évidemment, ce nouvel élément doit être greffé à la collection existante. Le sens n'est pas aléatoire contrairement à la création d'une règle. L'élément ajouté est d'office un esclave. Cependant, une telle mutation doit respecter la contrainte spécifiant le nombre de conditions maximum qu'une règle peut comporter. L'ajout d'une collection provoque la modification de l'ensemble des terminaux de la règle. De ce fait, le côté droit, directe-

ment dépendant de l'ensemble des terminaux de la règle, doit être "remanié" en prenant en compte ces nouveaux terminaux.

Supprimer une collection à gauche: Cette mutation est l'inverse de la précédente puisqu'elle enlève, au hasard, une des conditions (du côté gauche) de la règle. Afin de garder une règle syntaxiquement correcte, la suppression d'une collection va impliquer de nombreuses vérifications. Premièrement, il est nécessaire de respecter la contrainte spécifiant le nombre minimum de conditions avant d'effectuer une telle mutation. En effet, une règle ne peut se retrouver sans condition. Ensuite, il faut vérifier qu'il existe encore au moins une collection sans *NOT* (afin d'avoir des variables instanciées dans la partie gauche utilisables pour la partie droite) et au moins une collection définie à l'aide du méta-modèle source. Si l'ensemble des conditions restantes permet de garder une partie gauche valide, on peut alors supprimer tous les liens qui existent entre la collection supprimée et les autres collections et relier ces dernières entre-elles. Pour finir, si la règle utilisait des tests pour vérifier si une variable possède la valeur *NIL* ou des primitives de navigation, l'algorithme vérifie si ces tests restent compatibles. Sinon, ils sont simplement supprimés. De même que pour l'ajout d'une condition, cette mutation provoque une modification de l'ensemble des terminaux. Par contre, le remaniement du côté droit de la règle a lieu ici seulement si les faits utilisent des variables issues de la collection supprimée.

Recréer la partie gauche: Cette mutation consiste à faire tabula rasa de la partie gauche d'une règle, et de recréer aléatoirement cette partie gauche. Les terminaux ayant totalement disparus, il faut aussi effectuer un remaniement de la partie droite. Durant cette mutation, il faut veiller à créer une partie gauche respectant les bornes minimum et maximum du nombre de conditions.

Ajouter un fait à droite: Cette mutation consiste en l'ajout d'un fait, aléatoire, dans le côté droit de la règle. Il faut également veiller à ne pas dépasser la borne spécifiant le nombre maximum de faits dans le côté droit des règles. Ce nouvel ajout doit ensuite être lié avec les terminaux du côté gauche.

Supprimer un fait à droite: La suppression d'un fait est bien évidemment l'inverse de la mutation précédente puisqu'elle supprime un fait du coté droite de la règle. Ici aussi, le respect de la borne inférieure est requis. Cette mutation est la plus simple puisqu'aucune autre opération n'est requise après celle-ci.

Recréer la partie droite: Cette mutation est similaire à celle qui recrée le côté gauche, mais s'applique ici sur le côté droit. Les nouveaux faits

insérés doivent respecter les bornes inférieure et supérieure du nombre de faits pour le côté droit et doivent être liés avec les terminaux du côté gauche.

Changer les variables de la partie droite: Cette mutation permet de remanier aléatoirement les variables des faits du côté droit avec l'ensemble des terminaux du côté gauche. C'est en fait cette mutation qui est utilisée en complément de certaines des mutations présentées précédemment lorsque l'on parle du remaniement du côté droit. Cette mutation, contrairement aux autres, ne modifie par la structure des règles.

Mutations du contrôle Dans l'encodage des règles, en plus des conditions et faits impliqués dans celle-ci, figure deux entiers. Le premier, le numéro de module, permet d'attribuer un numéro de module à une règle. Le second est la priorité d'une règle. La modification de ces deux paramètres permet d'altérer le déroulement d'un programme et peut-être de l'améliorer en déclenchant plus tôt, ou plus tard, une règle. Deux mutations ont dès lors été imaginées:

Changement de module: Comme son nom l'indique, cette mutation assigne de manière aléatoire un numéro de module à une règle. Ce numéro est compris entre 0 et *ModuleMax*. *ModuleMax* est un paramètre du programme qui spécifie le nombre de modules maximum que le programme peut utiliser. Il est à noter qu'une telle mutation peut très bien attribuer le même numéro de module que précédemment à une règle. La mutation n'aura alors aucun effet.

Changement de la priorité: Le but de cette mutation est de modifier la priorité d'une règle. La règle se voit attribuer de manière aléatoire un numéro compris entre 0 et le nombre de règle total du programme. Ici aussi, cette mutation pourrait ne pas avoir d'effet si la règle reçoit le même numéro qu'avant l'appel à cette mutation.

Ces trois groupes de mutations (mutations du programme, des règles et du contrôle) sont mis en compétition, c'est-à-dire qu'un tirage au sort est effectué afin de savoir quel type de mutation sera effectué. Un deuxième tirage au sort permet ensuite de choisir l'une des mutations associées au type sélectionné.

Graphe

Un graphe a été implémenté et permet de visualiser, en temps réel, l'évolution d'une exécution. Celui-ci affiche la valeur de fitness (ou fonction objectif) et le nombre de règles du meilleur programme de chaque génération (fig. 4.25).

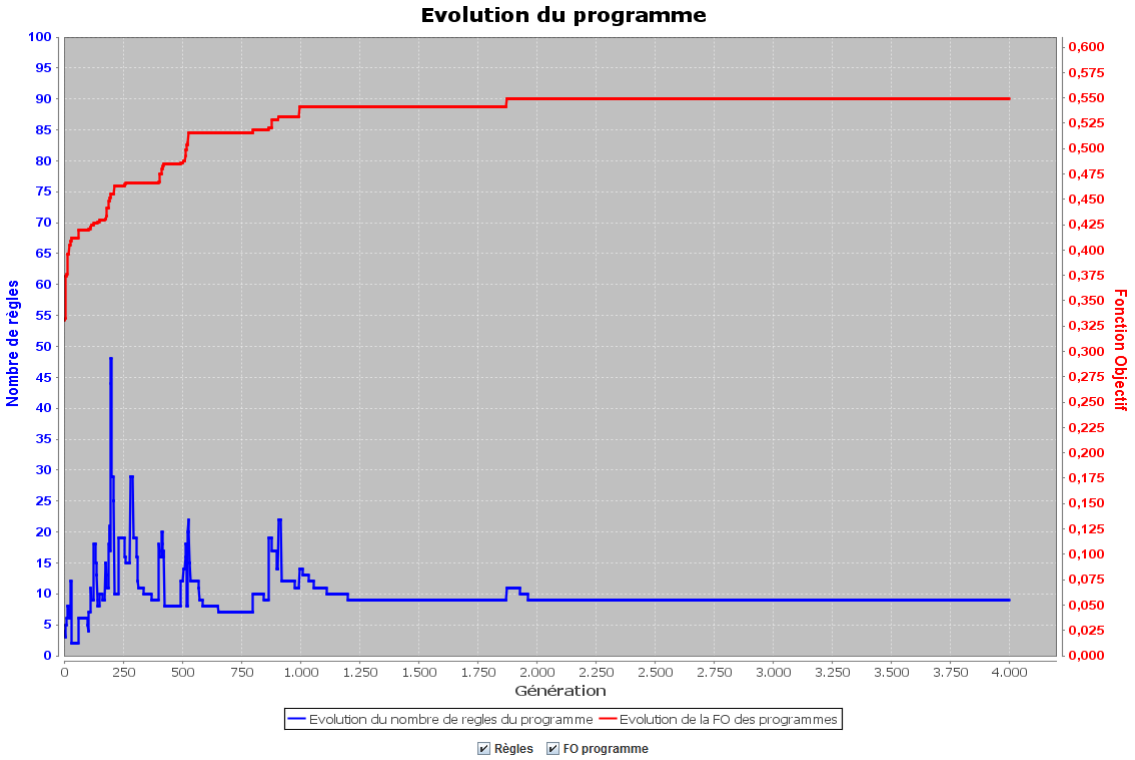

Figure 4.25: *Graphe d'exécution*

Création de l'ensemble des règles initiales

Les règles créées sont syntaxiquement correctes par rapport au langage de transformation (JESS) et cohérentes par rapport à leur méta-modèle.

Durant la phase consistant à générer la population initiale, les programmes sont constitués d'un certain nombre de règles compris entre *PRG_MIN_RULE* et *PRG_MAX_RULE* (où *PRG_MIN_RULE* et *PRG_MAX_RULE* sont des paramètres). Les règles créées contiennent un certain nombre de source patterns et target patterns. Ici aussi, quatre paramètres permettent de spécifier combien de source patterns et target patterns peuvent se trouver du côté gauche et combien de target patterns peuvent se trouver du côté droit (respectivement *MIN_LEFT_PAT*, *MAX_LEFT_PAT*, *MIN_RIGHT_PAT*, *MAX_RIGHT_PAT*). Il faut toutefois respecter le fait qu'une règle ne peut pas contenir que des target patterns du côté droit, et qu'au moins un source pattern doit être présent. Plus ces paramètres sont grands, plus l'espace de recherche augmente. Il faut donc les choisir avec rigueur.

Les différentes briques ont été identifiées et encodées. A la création d'une règle, le programme sélectionne au hasard les briques qui constitueront son côté gauche et les lie avec l'algorithme de liaison. De même, pour le côté droit de la règle, le programme sélectionne les faits qui constitueront l'action de la règle. Une fois le côté droit créé, ses variables doivent être liées avec les terminaux du côté gauche.

Évolution des règles et dérivation de nouveaux ensembles de règles

Grâce à la programmation génétique, la population de programmes va évoluer et donner lieu à des programmes de plus en plus performants grâce à l'utilisation des opérateurs génétiques de mutations et de croisement. Cependant, il faut veiller à ce que ces opérateurs produisent des règles syntaxiquement et sémantiquement correctes, tout comme la création initiale des règles.

Les opérateurs sont appliqués sur des programmes qui ont été sélectionnés en fonction de leur valeur de fitness. La méthode de sélection utilisée est la méthode de la roulette. Chaque ensemble de règle (programme) se voit assigner une probabilité d'être sélectionné qui est proportionnelle à la valeur de fitness de cet ensemble. Cette méthode est utilisée en raison de sa capacité à laisser une chance aux programmes ayant une faible valeur de fitness. Bien entendu, les programmes ayant une grande valeur de fitness ont plus de chance d'être sélectionnés que les autres. Mais parfois, un "mauvais programme" peut receler des éléments intéressants et il se peut qu'une légère amélioration le propulse dans l'ensemble des meilleurs programmes. Avec un peu de chance, cet individu amènera du nouveau dans le matériel génétique.

Un paramètre du programme permet de spécifier le nombre d'individus qui figureront dans l'élite, *ELITISM*. L'élitisme permet de sélectionner à chaque fois un certain nombre de bons programmes afin de garder ceux-ci dans les générations futures. Les programmes sont classés par ordre décroissant de valeur de fitness et les N meilleurs sont sélectionnés et directement insérés dans la nouvelle population. Ce paramètre doit également être choisi avec précaution. Un trop grand élitisme pourrait faire converger l'algorithme vers un programme non optimal, et un trop petit élitisme pourrait passer à côté de potentiellement bons programmes. Généralement, l'élitisme est fixé à un cinquième de la taille de la population.

Évaluation des ensembles de règles/programmes

Durant le processus évolutionnaire, chaque programme, c'est-à-dire chaque ensemble de règles est exécuté et évalué. Cette évaluation permet de donner une valeur associée à la performance des transformations opérées par chaque programme. Le programme est tout d'abord exécuté sur l'exemple

source fourni en entrée. Les règles de transformation produisent un nouveau modèle de même méta-modèle que le modèle cible fourni en entrée. Une comparaison est alors effectuée entre le modèle produit et le modèle cible fourni.

Les ensembles de règles sont préalablement traduit en JESS et exécutés grâce au moteur JESS. Les modèles sont encodés sous la forme d'ensemble de faits. A la fin de l'exécution, on récupère l'ensemble des faits générés et on le compare avec l'ensemble des faits du modèle cible fourni.

Il est important de signaler qu'un fait peut être asserté plusieurs fois. Cela n'a aucune conséquence puisque le fait sera à chaque fois écrasé. Par exemple, lorsqu'une classe C est impliquée dans plusieurs associations, il se peut qu'une règle de transformation chargée de transformer les associations crée autant de fois la table C que le nombre de fois où la classe C est impliquée dans une association. Cependant, rien n'empêche de veiller à ne pas créer inutilement des faits redondants.

La valeur attribuée au programme/ensemble de règles est la valeur de fitness ou encore la valeur de la fonction objectif. Cette valeur permet de mesurer la différence ou le taux d'erreur entre ce qui est produit par le programme et ce qui est attendu du programme. Autrement dit, on mesure ici la différence entre le modèle cible fourni et le modèle créé par le programme généré.

La méthode d'évaluation n'a pas changé par rapport à l'approche initiale. Dès lors, la fonction de fitness et la façon d'évaluer les programmes, décrites précédemment, restent identiques.

4.5.2 Approche RTC

Modifications relatives à la deuxième approche

Pour rappel, l'idée de la deuxième approche est de séparer le processus en deux phases. La première phase consiste en l'apprentissage des règles qui formeront le plan de transformation. Cependant, ces règles ne sont pas ordonnées. C'est en effet le rôle de la deuxième phase. Cette phase utilise l'algorithme du recuit simulé pour tenter de trouver un ordre d'exécution optimal pour l'ensemble des règles dérivées dans la première phase. Cependant, puisque les programmes de la première phase ne sont plus ordonnés, il n'est plus possible de les évaluer de la même façon que dans la première approche. Des modifications ont dû être apportées au niveau de l'encodage des règles afin de les évaluer une par une et non plus ensemble comme dans la première approche.

Encodage des règles

Dans cette deuxième approche, les règles contiennent toujours les mêmes propriétés que dans la première approche, à savoir un numéro de module, un numéro de priorité, un ensemble de briques qui forme la partie conditionnelle des règles et un ensemble de faits qui forme la partie action des règles. Dans cette autre approche, les règles contiennent également une valeur de fonction objectif qui leur est propre et qui permettra de leur attribuer un indice de performance. Cette valeur sera couplée avec celle des autres règles qui forment le programme pour déterminer la fonction objectif d'un programme dans cette deuxième approche.

Deux phases d'exécution

La deuxième approche est constituée de deux phases. La première phase fonctionne de manière analogue à la première approche puisqu'elle consiste à générer des programmes de transformation constitués de règles, qui évoluent avec des opérateurs génétiques. Cependant, ce processus est long. Lorsqu'il est question de générer 4000 générations de 400 programmes, le processus peut prendre jusqu'à 72 heures. La deuxième phase récupère une partie des programmes générés durant la première phase et procède à un ordonnancement de règles au moyen d'une heuristique appelée recuit simulé (Simulated Annealing en anglais). En fin de première phase, la population finale est "sérialisée", c'est-à-dire qu'elle est sauvegardée dans un fichier, qui pourra être "désérialisé" plus tard et être utilisé dans la deuxième phase. Cette fonctionnalité permet de ne pas avoir à exécuter les deux phases durant une même exécution, ce qui augmenterait encore le temps requis. De plus, cela permet d'obtenir des populations de programmes prêts à être ordonnancés et d'effectuer plusieurs fois la deuxième phase sur une même population, ce qui peut être très utile en phase de test. La valeur du paramètre *Phase* permet de configurer le déroulement d'une exécution:

- 0 : Exécuter les phase 1 puis la phase 2.
- 1 : Exécuter uniquement la phase 1.
- 2 : Exécuter uniquement la phase 2.

Les valeurs 0 et 2 nécessitent la spécification de l'emplacement du fichier contenant la sérialisation de la population à ordonner.

Le recuit simulé

Le recuit simulé est une méta-heuristique probabiliste tout droit venue de la métallurgie. "Recuit simulé" est la traduction de Simulated Annealing. Dans la réalité, le terme "recuit" correspond à un réchauffement suivi

d'un refroidissement lent. Selon que des métaux chauffés au rouge sont refroidis rapidement ou lentement, ils présentent des propriétés complètement différentes. Lorsque l'on souhaite obtenir un métal parfait (cristal) c'est-à-dire dont l'énergie interne est à son minimum, il est nécessaire d'appliquer le processus du recuit. Après avoir atteint la liquéfaction du métal, il est nécessaire de le refroidir en diminuant la température. Cependant, c'est un processus lent, car si la température descend trop brusquement, le métal sera de type "verre" et comportera alors des défauts. En revanche, si la chute de température est lente et maîtrisée, la structure interne du métal sera alors de plus en plus proche de celle du cristal. Dans le cas d'une chute de température trop rapide, il est possible de gommer certains défauts en réchauffant le métal afin que les atomes se replacent mieux. D'où le nom de recuit. On alterne les cycles de refroidissement lent et de réchauffage qui ont pour effet de minimiser l'énergie du matériau. Les qualités du matériau sont améliorées, tandis que ses défauts sont atténués. Cette méthode est transposée en optimisation pour trouver les extrema d'une fonction par trois chercheurs de la société IBM, S. Kirkpatrick, C.D. Gelatt et M.P. Vecchi en 1983 [51], et indépendamment par V. Černy en 1985 [101].

Le recuit simulé est utilisé dans de nombreux domaines dans lesquels on a à résoudre des problèmes d'optimisation difficiles. Il permet de trouver une bonne approximation de l'optimum global d'une fonction donnée dans un espace de recherche de grande taille. Il est la plupart du temps utilisé pour une recherche dans un espace d'états discrets. Le but n'étant pas de trouver LA meilleure solution, il est utilisé lorsque l'on souhaite trouver une solution acceptable, ce qui rend la recherche moins longue qu'un énumération complète de l'espace de recherche.

Contrairement à d'autres heuristiques, il a pu être prouvé que la méthode du recuit simulé convergeait vers un optimum global et non local. Cela veut donc dire que le recuit simulé peut trouver la meilleure solution si on le laisse chercher indéfiniment.

Le recuit simulé améliore la recherche grâce à deux choses:

1. L'algorithme de Metropolis [68] permet d'accepter des solutions qui ne diminuent pas forcément l'énergie du système. Ce premier point permet en fait d'explorer l'espace de recherche plus en profondeur, au contraire d'autres heuristiques.

2. En analogie avec la métallurgie, la diminution de la température permet de limiter le nombre de moins bonnes solutions au fil du temps. Une fois que la température a atteint un certain palier, l'heuristique n'a alors tendance qu'à accepter seulement les bonnes solutions.

Certaines méthodes d'optimisation ("greedy") peuvent être dépendantes du point de départ, et dès lors rester bloquées dans sur un minima local.

Algorithme

Plusieurs algorithmes du recuit simulé sont disponibles [82]. Celui utilisé dans l'approche RTC se présente comme suit:

```
Sélectionner une solution initiale S_0
Sélectionner une température t_0 > 0
Sélectionner un réducteur de température 0 < α < 1
Répéter
        Répéter
                Choisir un voisin S de S_0 (Perturber S_0)
                δ = FO(S) − FO(S_0)
                Si δ > 0 alors S_0 = S
                Sinon
                        Sélectionner un x entre 0 et 1
                        Si x < e^(δ/t) alors S_0 = S
        Tant que #itération < nrep
        t = t ∗ α
Tant que t > 0.01
S_0 est la meilleur solution
```

État initial de l'algorithme La solution initiale S_0 peut être prise au hasard dans l'espace des solutions possibles. À cette solution correspond une énergie initiale $FO(S_0)$. Une température initiale t_0 élevée est également choisie. Ce choix est alors totalement arbitraire.

Itérations de l'algorithme À chaque itération de l'algorithme, la solution sera modifié, on dira qu'elle subit une "perturbation". Cette modification entraîne une variation de l'énergie du système. Si cette variation est positive alors il s'agit d'une meilleure solution puisqu'elle fait augmenter l'énergie du système. Sinon, elle est acceptée avec une probabilité $e^{\frac{\delta}{t}}$. Une dégradation faible est acceptée avec une probabilité plus grande qu'une dégradation plus importante.

L'idée est d'effectuer un mouvement selon une distribution de probabilité qui dépend de la qualité des différents voisins:

- Les meilleurs voisins ont une probabilité plus élevée.
- Les moins bons ont une probabilité plus faible.

Il existe deux approches standards quant à la variation de la température:

1. Pour la première, on itère en gardant la température constante. Lorsque le système a atteint un équilibre thermodynamique, on diminue la température du système. On parle alors de paliers de température.
2. La seconde approche fait baisser la température de façon continue. On peut opter pour une décroissance géométrique consistant à utiliser un paramètre, disons α, compris entre 0.9 et 0.99. A chaque itération, on multiplie la température courante par α. Si le refroidissement est trop rapide, il y a un risque de rester bloqué dans un minimum local (configuration sous-optimale).

On peut aussi utiliser d'autres schémas de refroidissement, où on utilise parfois une température constante ou bien on peut utiliser des schémas plus complexes, dans lesquels la température peut parfois remonter.

On voit bien que la température à un rôle crucial dans l'algorithme. Lorsque celle-ci est haute, le système peut se déplacer dans l'espace des solutions ($e^{\frac{\delta}{t}}$ proche de 1) en choisissant des solutions moins bonnes que la meilleure solution actuelle. En revanche, lorsque la température est basse, les solutions augmentant l'énergie du système sont bien sûr choisies, mais d'autres peuvent être acceptées (de façon restreinte cependant), empêchant ainsi l'algorithme de tomber dans un minimum local.

Paramètres Les principaux inconvénients du recuit simulé résident dans le choix des nombreux paramètres de l'algorithme. Ces paramètres sont souvent choisis de manière empirique:

La solution initiale S0

La solution initiale consiste en un ensemble de règles provenant d'un pool de programmes (préalablement paramétré par l'utilisateur). Bien entendu, les programmes constituant ce pool sont sélectionnés selon leur valeur de fonction objectif (décroissante).

La température

La température est un des paramètres permettant de contrôler la durée de l'algorithme. Durant un palier de température, des modifications de l'ordre d'exécution auront lieu un nombre arbitraire de fois ($nrep$).

Réducteur de température

Le réducteur de température est un nombre spécifiant la façon dont la température va décroître. En générale, on utilise un nombre compris entre 0 et 1, le plus souvent proche de 0.9. Plus ce nombre est proche de

1, plus l'algorithme durera longtemps. Il faut savoir qu'au début, lorsque la température est encore assez haute, les solutions dégradant la meilleure solution ont plus de chance d'être sélectionnée étant donné $e^{\frac{\delta}{t}}$. A la fin, c'est l'inverse. Il faut donc choisir avec minutie la valeur du réducteur de température pour ne pas aller trop vite, ou trop lentement.

Critère d'arrêt

Plusieurs critères d'arrêt sont possibles. On peut décider de s'arrêter après un certain nombre d'itération, si l'énergie est en dessous d'un seuil fixé, si l'énergie ne varie plus assez, si la température est en dessous d'un certain seuil, etc.

Fonction de pertubation Une solution consiste en un ensemble de règles ordonnées. L'espace de recherche contient M^N solutions où N représente le nombre de règle et M le nombre de module. La fonction de perturbation permet de visiter le voisinage d'une solution. En perturbant une solution, on change l'ordre des règles et on arrive sur un voisin de la solution. La fonction de perturbation change les modules associés aux règles et cela donne une nouvelle solution qui sera meilleure, moins bonne ou équivalente en terme d'ordonnancement des règles.

Cependant, il se peut que des règles inutiles soient introduites dans la solution initiale, dû au fait qu'on sélectionne des règles provenant d'un ensemble de programmes. Il est donc nécessaire de s'en débarrasser afin qu'elles ne détériorent pas la solution finale. JESS nous permet de définir des modules, d'y placer des règles dedans et de les placer dans une pile afin de les exécuter. Mais il ne nous est pas interdit de déclarer un module, d'y placer des règles dedans et de ne pas le mettre dans la pile, de façon à ne pas avoir à exécuter les règles y étant insérées ! Ce module "poubelle" devrait contenir, à la fin de l'exécution, les règles inutiles ou détériorant la qualité du programme.

Évaluation des programmes

L'évaluation des programmes dans cette deuxième approche fonctionne différemment que dans l'approche RAC. Vu que les programmes ne sont plus ordonnés dans la première phase, on ne dispose plus que d'un simple ensemble de règles. On ne peut donc plus évaluer ces programmes de la même façon que dans la première approche. En revanche, on peut tenter d'évaluer la capacité d'un ensemble de règles à pourvoir transformer un modèle en un autre en calculant d'une part, la performance individuelle des règles et d'autre part, la couverture fournie par l'ensemble des règles.

Évaluation des règles d'un programme (Phase 1) Une règle possède désormais une nouvelle propriété qui est sa fonction objectif. Celle-ci mesure sa capacité à créer de bons faits (Full matches). Elle est déterminée par:

$$FOr = \frac{\#Full\ matches\ créés\ par\ la\ règle}{\#faits\ créés\ par\ la\ règle}$$

Cette valeur varie entre 0 et 1 et plus elle sera proche de 1, plus la règle sera parfaite et adéquate à la transformation du modèle source. On dira que la règle ne crée pas de bruit.

Il faut ensuite faire une moyenne de ces fonctions objectifs, ce qui donnera la fonction objectif moyenne des règles et qui est déterminée par:

$$FOMoyr = \sum \frac{FOr}{\#règles}$$

Cependant cette valeur seule ne suffit pas. En effet, on imagine très bien le cas d'un programme produisant une seule règle, mais parfaite. La fonction objectif globale du programme serait donc de 100%. Il faut donc calculer la couverture de l'ensemble des règles constituant le programme:

$$Couverture = \frac{\#Full\ matches\ créés\ par\ le\ programme}{\#Faits\ du\ modèle\ cible}$$

Dès lors, on peut combiner ces deux valeurs pour calculer la fonction objectif globale d'un programme:

$$FOp = \alpha * FOMoyr + \beta * Couverture \qquad où\ \ \alpha + \beta = 1$$

Si FOp vaut 1, alors le programme couvre tous les faits attendus et ne crée pas de bruit.

Cette évaluation n'est pas encore parfaite car il reste encore deux autres problèmes à régler.

Le premier problème vient du fait que chaque règle est testée séparément, alors que justement, dans un programme de transformation, certaines règles ont besoin de la production des actions d'autres règles pour se déclencher. Une règle qui possède une collection définie à l'aide du méta-modèle cible (target pattern) dans sa partie gauche ne pourra jamais être déclenchée et donc être évaluée. La fonction objectif de ces règles sera mise à zéro et elles seront directement écartées.

Pour résoudre ce problème, lorsqu'on souhaite évaluer une règle, les faits du modèle cible qu'on souhaite récolter sont introduits en entrée avec ceux du modèle source. Afin de différencier les faits injectés des nouveaux faits produits par la règle qui est évaluée, un préfixe est rajouté aux faits injectés. Les règles peuvent désormais être évaluées de manière totalement

indépendante.

Le deuxième problème qui est vite apparu est qu'il est possible de générer des doublons ou des règles qui produisent les mêmes faits. Le fait de négliger la redondance de ces règles permettrait de générer des programmes gigantesques qui pourraient contenir plusieurs milliers de règles après à peine quelques centaines de générations. Comme pour les lois de la nature, si une espèce ne rencontre aucune contrainte pour se reproduire, sa croissance aura tendance à devenir fortement problématique à un moment donné. La redondance d'un fait se calcule comme suit:

$$RedondanceFait = 1 - \frac{\#répétition}{borneRedondance}$$

La *borneRedondance* est un paramètre spécifié par l'utilisateur. Une fois ce nombre dépassé, la redondance sera d'office maximale. Il faut alors calculer la redondance moyenne des faits d'un programme:

$$RedondanceProgramme = \frac{\sum RedondanceFait}{\#Faits\,assertés}$$

Une nouvelle contrainte a donc été mise en place dans la fonction objectif:

$$FOp = \alpha * FOMoyr + \beta * Couverture + \gamma * RedondanceProgramme$$

$$\text{où } \alpha + \beta + \gamma = 1$$

Evaluation des programmes (Phase 2) En phase 2, la méthode d'évaluation de l'approche RAC convient, puisqu'on cherche à évaluer un ordonnancement de règles dans un programme.

Graphes

Tout comme pour la première approche, un graphique permet de suivre l'évolution en temps réel du meilleur programme durant la première phase. Au nombre de règles et à la fonction objectif globale, s'ajoutent également la fonction objectif moyenne des règles et la couverture du meilleur programme (fig. 4.26).

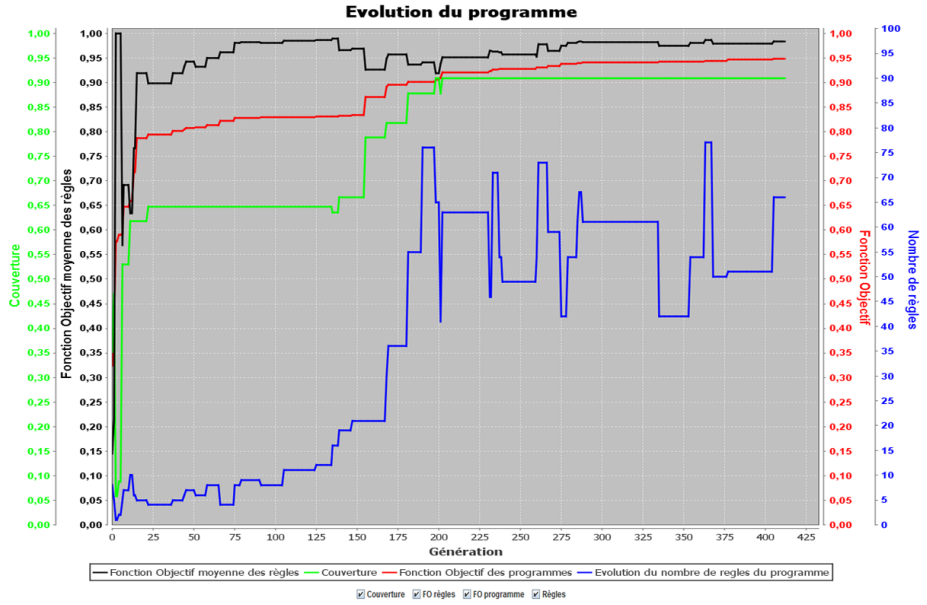

Figure 4.26: *Graphe d'exécution (Phase 1)*

Un graphe a également été utilisé pour monitorer l'exécution durant la deuxième phase, où l'on peut voir l'évolution de la température, du nombre de règles et de la valeur de la fonction objectif(fig. 4.27).

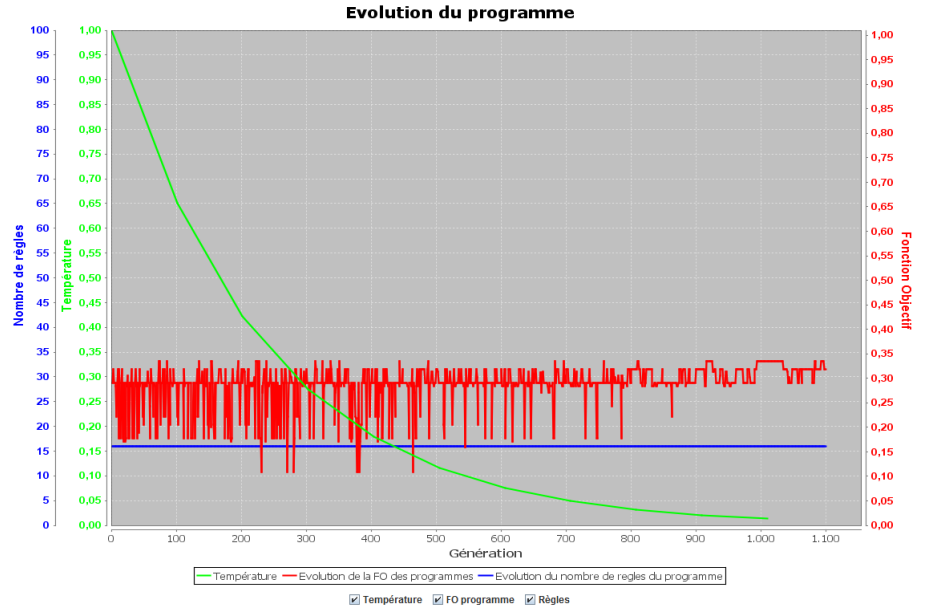

Figure 4.27: *Graphe d'exécution (Phase 2)*

Chapitre 5

Étude de cas

5.1 Description

L'application a pour objectif de générer un programme qui permet de transformer un modèle de méta-modèle *A* en un modèle correspondant de méta-modèle *B*. Dans la situation précédente, l'application avait été testée sur deux types de transformation de modèles: la transformation de diagrammes de classes vers des schémas relationnels ainsi que des machines à états vers des diagrammes de séquences. Par contre, l'étude de cas proposée dans ce mémoire a été réalisée sur un seul type de transformation de modèles: la transformation de diagrammes de classe en schémas relationnels. Aussi, cette étude de cas se limitera à l'approche RAC. En effet, l'approche RTC n'a pas été suffisamment testée pour être évaluée de la même manière que l'approche RAC. Cette étude de cas est composée en cinq étapes. Nous présenterons d'abord le plan de transformation diagramme de classe vers schéma relationnel idéal et comment celui-ci est représenté en langage JESS. Nous analyserons la structure de ce programme de transformation afin de comprendre l'ordonnancement des règles. Nous verrons également les difficultés liées à ce programme idéal. Une évaluation qualitative sera présentée afin de comparer les règles générées durant une exécution avec celles du plan de transformation idéal. Étant donné que les programmes possèdent une fonction objectif, une évaluation quantitative permettra de voir, d'une part, si les résultats sont constants et, d'autre part, si ceux-ci ont été améliorés par rapport à la solution initiale. Un testeur de règles a également été implémenté. Celui-ci a pour but de vérifier que les règles issues du programme idéal soient bien générables. Le détail de son fonctionnement sera présenté. Enfin, l'étude se terminera par une petite discussion.

5.2 Programme idéal

La première étape avant de vouloir générer des plans de transformation étaient de créer à la main la solution idéale (fig. 5.1). La solution recherchée devait pouvoir transformer n'importe quel exemple de diagramme de classe en schéma relationnel correspondant. Plusieurs versions ont été écrites afin de trouver la plus optimisée au niveau du nombre de règles et de leur complexité. Le but était de savoir s'il était possible de créer un plan pouvant être généré à partir de règles qui respectaient la structure imposée par notre application. C'est-à-dire, comme présenté auparavant, une règle doit contenir dans sa partie gauche des briques préfabriquées reliées entre elles et des faits reliés entre-eux par une propriété dans sa partie droite.

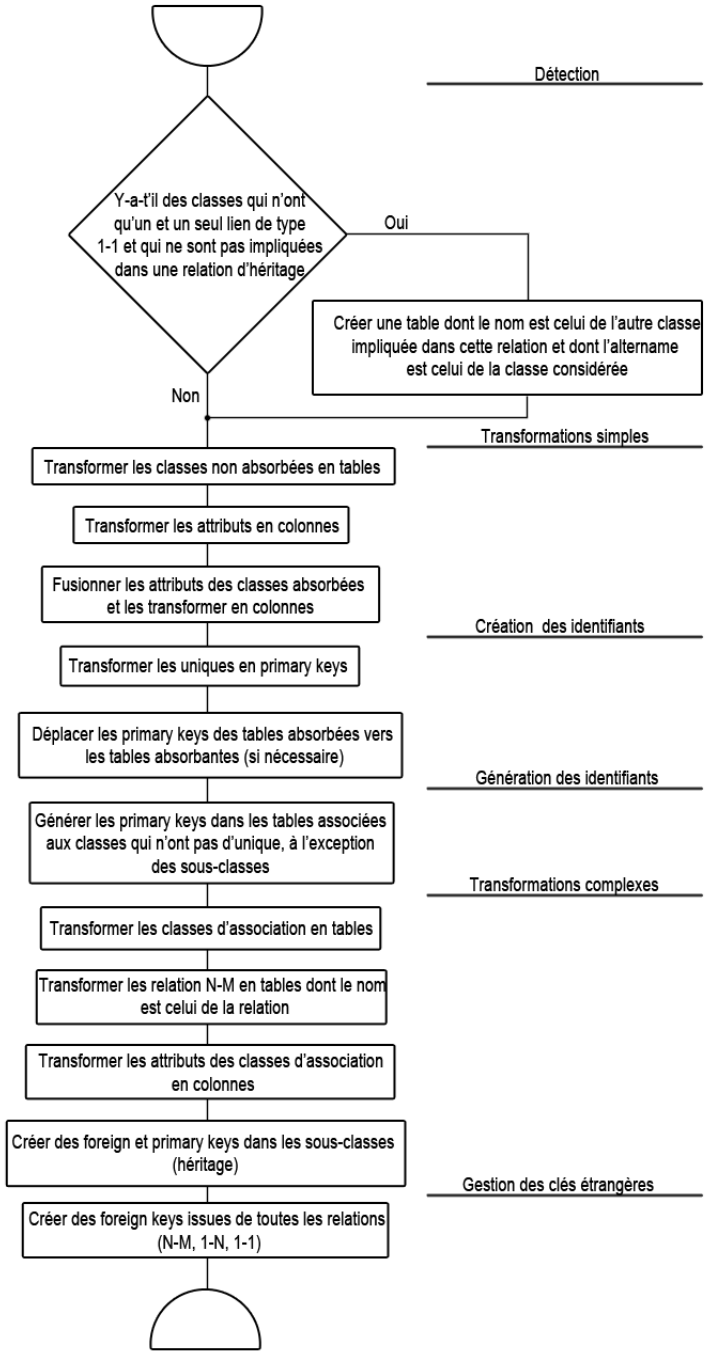

Figure 5.1: *Programme de transformation de diagrammes de classe en schémas relationnels idéal*

Un plan de transformation généré en JESS par l'application possède une structure bien définie:

```
(clear)
;Templates
...
;Querys
...
;Asserts
...
;Modules
...
(focus module1 ... moduleN)
(run)
```

Premièrement, la commande *(clear)* va supprimer tous les faits présents dans la mémoire de travail pour ne pas garder des traces d'exécution des programmes évalués précédemment par l'application.

Deuxièmement, les méta-modèles sources et cibles sont déclarés à l'aide de templates en JESS. Les méta-modèles sont prédéfinis dans l'application et sont juste copiés à chaque fois dans les programmes générés.

```
;Templates

(deftemplate class (slot name(type String)))
(deftemplate attribute (slot name(type String))(slot class(type String))(slot unico(
    type Integer)(default 0)))
(deftemplate association1n (slot classfr(type String))(slot classto(type String)))
(deftemplate associationnm (slot name(type String))(slot classfr(type String))(slot
    classto(type String))(slot classas(type String)));
(deftemplate association11 (slot classa(type String))(slot classb(type String)))
(deftemplate inheritance (slot class(type String))(slot superclass(type String)))

(deftemplate table (slot name(type String))(slot alternname(type String)(default nil))
    )
(deftemplate column (slot name(type String))(slot table(type String))(slot pk(type
    Integer)))
(deftemplate fk (slot column(type String))(slot table(type String))(slot fktable(type
    String)))
```

Ensuite, ce sont les primitives de navigations (*Querys*) qui sont déclarées afin de pouvoir être utilisées par les règles de transformation. Comme les méta-modèles, les primitives sont des objets préfabriqués à l'avance.

```
;Querys

(defquery allAssociation
 (declare (variables ?c))
 (or (association11 (classb ?c))
 (or (association11 (classa ?c))
 (or (association1n (classfr ?c))
 (or (association1n (classto ?c))
 (or (associationnm (classfr ?c))(associationnm (classto ?c)))))))
)
```

Après cela, les faits (*Asserts*) du modèle source sont insérés dans la mémoire de travail.

Enfin, après toutes ces déclarations, vient le corps du programme. Il est composé de plusieurs modules dans lesquels sont placés les différentes règles générées par notre application et traduites en JESS. Le plan de transformation idéal est constitué de six modules. Chaque module possède des règles comportant au plus trois briques dans leur partie gauche et deux asserts dans leur partie droite. L'ensemble des modules totalise 20 règles de transformation.

Le premier module, *Detection*, possède deux règles qui ont pour but de détecter si deux classes peuvent être fusionnées en une table. Pour rappel, il faut que les deux classes soient reliées par une association 1-1 et que l'une des deux classes ne possède pas d'autres associations et qu'elle ne possède pas de sous-classes.

On peut constater que la description d'une association 1-1 dans le méta-modèle à un sens. Pour tester tous les cas, il faut deux règles car il faut vérifier si c'est la classe source qui peut être fusionnée dans la classe cible ou bien l'inverse.

```
(defmodule Detection);

(defrule Detect−Class−Assoc−11
 (association11 (classa ?c00)(classb ?c10 ))
 (class(name ?c00))
 (class(name ?c10))

 (not (inheritance (class ?c20)(superclass ?c00)))
 (class(name ?c00))

 (test (eq (count−query−results allAssociation ?c00) 1))
 =>
 (assert (table (name ?c10)(alternname ?c00)))
)

(defrule Detect−Class−Assoc−11bis
 (association11 (classa ?c00)(classb ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (not (inheritance (class ?c20)(superclass ?c10)))
 (class(name ?c10))

 (test (eq (count−query−results allAssociation ?c10) 1))
 =>
 (assert (table (name ?c00)(alternname ?c10)))
)
```

Le deuxième module, *TransfoSimples*, possède quatre règles. La première va transformer toutes les classes qui n'ont pas pu être fusionnées en une table. La seconde va transformer tous les attributs non unique d'une classe A en colonne pour les tables A correspondantes qui ont déjà été créées. Les deux dernières vont transférer et transformer tous les attributs d'une classe B fusionnée avec une classe A en colonne pour les tables A correspondantes qui ont déjà été créées. La première fusionne les attributs à condition que la classe A possède au moins un attribut unique. Cette condition permet d'éviter l'ajout de clé primaire en relation avec les attributs unique de la classe A absorbée. Tandis que la deuxième fusionne les attributs dans un contexte où la classe A ne possède pas d'attribut unique. Dans ce cas, les attributs uniques de la classe B pourront devenir les clés primaires dans la table résultante.

```
(defmodule TransfoSimples);

(defrule Transfo−Class−2−Table
 (class (name ?c00))

 (not (table (name ?t20)(alternename ?c00)))

 (not (table (name ?c00)(alternename ?t41)))
 =>
 (assert (table (name ?c00)(alternename nil)))
)

(defrule Transfo−Attribute−2−Column
 (attribute (name ?a00)(class ?c00)(unico 0))
 (class(name ?c00))

 (table (name ?c00)(alternename ?t21))
 =>
 (assert (column (name ?a00)(table ?c00)(pk 0)))
)

(defrule Transfo−Merged−Attribute−Column
 (attribute (name ?a00)(class ?c00)(unico ?a02))
 (class(name ?c00))

 (table (name ?t20)(alternename ?c00))

 (attribute (name ?a40)(class ?t20)(unico 1))
 (class(name ?t20))
 =>
 (assert (column (name ?a00)(table ?t20)(pk 0)))
)

(defrule Transfo−Merged−Attribute−Column2
 (attribute (name ?a00)(class ?c00)(unico ?a02))
 (class(name ?c00))

 (table (name ?t20)(alternename ?c00))

 (not (attribute (name ?a40)(class ?t20)(unico 1)))
 (class(name ?t20))
 =>
 (assert (column (name ?a00)(table ?t20)(pk ?a02)))
)
```

Le troisième module, *CréationPK*, permet de transformer les attributs uniques en clés primaires. Il possède deux règles distinctes. La première règle va transformer les attributs uniques d'une classe A qui a été absorbée lors d'une fusion avec une classe B en clé primaire dans la table B si la classe B ne possède pas d'attribut unique. La dernière règle transforme simplement les attributs uniques d'une classe non absorbée en clé primaire dans la table correspondante.

```
(defmodule CreationID);

(defrule Change-PK-after-fusion
 (attribute (name ?a00)(class ?c00)(unico 1))
 (class(name ?c00))

 (table (name ?t20)(alternname ?c00))

 (not (attribute (name ?a40)(class ?t20)(unico 1)))
 (class(name ?t20))
 =>
 (assert (column (name ?a00)(table ?t20)(pk 1)))
)

(defrule Transfo-Unico-2-PK
 (attribute (name ?a00)(class ?c00)(unico 1))
 (class(name ?c00))

 (table (name ?c00)(alternname ?t21))
 =>
 (assert (column (name ?a00)(table ?c00)(pk 1)))
)
```

Le quatrième module, *GenerationPK*, permet de générer des identifiants techniques pour les tables qui n'ont pas reçu de clé primaire durant l'exécution des modules précédents. Il est constitué d'une règle qui vérifie s'il existe une table qui ne possède pas d'identifiant ni de super-classe. Cette dernière condition est nécessaire car l'héritage n'a pas encore été traité durant les modules précédents. Cet ordre a été choisi afin de garantir que, si une sous-classe ne possède pas identifiant, elle recevra l'identifiant de sa super-classe. Si le modèle source possède une chaîne de classes reliées par des relations d'héritage, et si ses classes ne possèdent pas d'identifiants, seule la classe au sommet de cette chaîne recevra un identifiant technique. Les autres recevront cet identifiant lors de la création des clés étrangères liées aux relations d'héritages.

```
(defmodule GenerationPK);

(defrule Gen−PK
 (table (name ?t00)(alternname ?t01))

 (not (column (name ?o20)(table ?t00)(pk 1)))
 (table (name ?t00)(alternname ?t21))

 (not (inheritance (class ?t00)(superclass ?c50)))
 (class(name ?t00))
 =>
 (assert (column (name ?t00)(table ?t00)(pk 1)))
)
```

Le cinquième module, *TransfoComplexes*, va s'occuper essentiellement de la transformation des classes d'associations. La première règle, *Transfo-ClassAs-Attribute*, transfère tous les attributs des classes d'associations en colonnes des tables d'associations correspondantes. La seconde règle, *Transfo-ClassAs-2-Table*, transforme les classes d'associations en tables d'associations. La troisième règle, *Transfo-Renamed-ClassAs-2-Table* transforme les associations N-M ne possédant pas de classe d'association en table d'association en utilisant le nom de leur rôle. Enfin, la dernière règle, **Inheritance**, ajoute une clé étrangère reliant les super-classes et les sous-classes.

```
(defmodule TransfoComplexes)

(defrule Transfo−ClassAs−Attribute
 (associationnm (name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
 (class(name ?c00))
 (class(name ?c10))

 (attribute (name ?a20)(class ?s03)(unico ?a22))
 (class(name ?s03))
 =>
 (assert (column (name ?a20) (table ?s03)(pk ?a22)))
)

(defrule Transfo−ClassAs−2−Table
 (associationnm (name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
 (class(name ?c00))
 (class(name ?c10))

 (test (neq ?s03 nil))
 =>
 (assert (table (name ?s03)(alternname nil)))
)

(defrule Transfo−Renamed−ClassAs−2−Table
 (associationnm (name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
```

```
 (class(name ?c00))
 (class(name ?c10))

 (test (eq ?s03 nil))
 =>
 (assert (table (name ?s00)(alternname nil)))
)

(defrule Inheritance
 (inheritance (class ?c00)(superclass ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (column (name ?o20)(table ?c10)(pk 1))
 (table (name ?c10)(alternname ?t21))
 =>
 (assert (fk(column ?o20)(table ?c00)(fktable ?c10)))
 (assert (column(name ?o20)(table ?c00)(pk 1)))
)
```

Le sixième module, *GenerationFK*, s'occupe essentiellement de l'ajout des clés étrangères. Lorsque nous avons affaire à des associations de type 1-1 ou de type N-M entre deux classes A et B, les tables A et B auront chacune une clé étrangère qui référence l'autre table. Le module possède donc pour chacune de ces transformations deux règles miroirs. L'une s'occupant d'ajouter la clé étrangère dans la classe A et l'autre dans la classe B.

Les deux premières règles, *Create-FK-11* et *Create-FK-11b*, rajoutent des clés étrangères dans les tables dont les classes d'origines sont reliées par une association 1-1. La deuxième paire de règles, composée de *Create-FK-nm* et *Create-FK-nm2*, s'occupe de rajouter des clés étrangères dans les tables d'associations pointant vers les tables A et B associées.

La troisième paire de règles, qui contient *Create-FK-Renamed-nm* et *Create-FK-Renamed-nm2*, exécute le même travail que la paire précédente à l'exception du fait que les tables d'associations traitées ont été générées à l'aide du nom du rôle des associations N-M ne possédant pas de classes d'associations. La dernière règle, *Create-FK-1n*, ajoute une clé étrangère entre deux tables dont leurs classes respectives étaient reliées par une association 1-N.

```
(defmodule GenerationFK);

(defrule Create-FK-11
 (association11 (classa ?c00) (classb ?c10))
 (class(name ?c00))
 (class(name ?c10))
```

```
 (column (name ?o20)(table ?c00)(pk 1))
 (table (name ?c00)(altername ?t21))

 (test (eq ?t21 nil))
 =>
 (assert (fk (column ?o20) (table ?c10) (fktable ?c00)))
 (assert (column(name ?o20)(table ?c10)(pk 0)))
)

(defrule Create-FK-11b
 (association11 (classa ?c00) (classb ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (column (name ?o20)(table ?c10)(pk 1))
 (table (name ?c10)(altername ?t21))

 (test (eq ?t21 nil))
 =>
 (assert (fk (column ?o20) (table ?c00) (fktable ?c10)))
 (assert (column(name ?o20)(table ?c00)(pk 0)))
)

(defrule Create-FK-nm
 (associationnm (name ?s00)(classfr ?c00) (classto ?c10) (classas ?s03))
 (class(name ?c00))
 (class(name ?c10))

 (column (name ?o20)(table ?c00)(pk 1))
 (table (name ?c00)(altername ?t21))

 (test (neq ?s03 nil))
 =>
 (assert (fk (column ?o20) (table ?s03) (fktable ?c00)))
 (assert (column (name ?o20)(table ?s03)(pk 1)))
)

(defrule Create-FK-nm2
 (associationnm(name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
 (class(name ?c00))
 (class(name ?c10))

 (column(name ?o20)(table ?c10)(pk 1))
 (table(name ?c10)(altername ?t21))

 (test (neq ?s03 nil))
 =>
 (assert (fk(column ?o20)(table ?s03)(fktable ?c10)))
 (assert (column (name ?o20)(table ?s03)(pk 1)))
)
(defrule Create-FK-Renamed-nm
 (associationnm(name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
 (class(name ?c00))
```

```
 (class(name ?c10))

 (column(name ?o20)(table ?c00)(pk 1))
 (table(name ?c00)(altername ?t21))

 (test (eq ?s03 nil))
 =>
 (assert (fk(column ?o20)(table ?s00)(fktable ?c00)))
 (assert (column (name ?o20)(table ?s00)(pk 1)))
)

(defrule Create-FK-Renamed-nm2
 (associationnm(name ?s00)(classfr ?c00)(classto ?c10)(classas ?s03))
 (class(name ?c00))
 (class(name ?c10))

 (column(name ?o20)(table ?c10)(pk 1))
 (table(name ?c10)(altername ?t21))

 (test (eq ?s03 nil))
 =>
 (assert (fk(column ?o20)(table ?s00)(fktable ?c10)))
 (assert (column (name ?o20)(table ?s00)(pk 1)))
)

(defrule Create-FK-1n
 (association1n (classfr ?c00) (classto ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (column (name ?o20)(table ?c00)(pk 1))
 (table (name ?c00)(altername ?t21))
 =>
 (assert (fk (column ?o20) (table ?c10) (fktable ?c00)))
 (assert (column (name ?o20)(table ?c10)(pk 0)))
)
```

Pour finir, la pile d'exécution des modules est déclarée afin de connaître l'ordre d'exécution de ces derniers. La commande *(run)* démarre le moteur d'inférence.

```
;Gestion stack
(focus Detection TransfoSimples CreationPK GenerationPK TransfoComplexes
    GenerationFK)

(run)
```

5.3 Évaluation qualitative

Cette analyse qualitative va nous permettre de comprendre pourquoi certaines règles ont été générées de manière imprévisible.

Il est à constater que les programmes générés possédant 100% de fonction objectif ne correspondaient pas forcément au plan de transformation idéal ou à l'un de ses sous-ensembles.

Les exemples de modèles de diagrammes de classe et de schémas relationnels fournis en entrée ne possèdent pas tous les cas particuliers qu'il est possible de rencontrer lors d'une transformation d'un modèle de diagramme de classe en schéma relationnel. Ces paires d'exemple ont, en général, pour effet de produire soit des règles ressemblant à celles souhaitées mais de nature moins "générale", soit des règles totalement différentes. On peut aussi constater que les exemples de modèles avec un haut niveau de complexité permettent de générer des programmes qui ressemblent beaucoup plus au programme idéal alors que leur fonction objective n'atteint pas les 100%.

Règle générée:

```
(defrule R_23348251
 (association11(classa ?c00)(classb ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (attribute(name ?a20)(class ?c10)(unico 1))
 (class(name ?c10))

 =>
 (assert (fk(column ?a20)(table ?c00)(fktable ?c10)))
 (assert (column(name ?a20)(table ?c00)(pk 0)))
)
```

Règle idéale:

```
(defrule Create-FK-11b
 (association11 (classa ?c00) (classb ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (column (name ?o20)(table ?c10)(pk 1))
 (table (name ?c10)(alternname ?t21))

 (test (eq ?t21 nil))
 =>
 (assert (fk (column ?o20) (table ?c00) (fktable ?c10)))
 (assert (column(name ?o20)(table ?c00)(pk 0)))
)
```

Voici un exemple typique de règle générée lorsqu'un modèle de diagramme de classe passé en argument ne possède pas de construction permettant de fusionner certaines classes. La règle générée va simplement vérifier que la classe $c10$ possède un attribut unique pour ajouter une clé étrangère entre la table $c00$ et $c10$. Vu que le modèle ne fournit pas de contre-exemple (comme par exemple dans le cas de deux classes qui fusionnent, il faut éviter de créer une clé étrangère entre la table résultante et la table qui n'existe pas à cause de la fusion), la règle générée est dans ce cas-ci satisfaisante. Étant moins complexe que la règle idéale et ne fournissant pas de *partial-match* ni de *no-match*, elle possède une plus forte probabilité d'être générée à la place de la règle attendue.

Règle générée:

```
(defrule R_23348246
 (attribute(name ?a00)(class ?c00)(unico 1))
 (class(name ?c00))

 (association1n(classfr ?c00)(classto ?c30))
 (class(name ?c00))
 (class(name ?c30))
 =>
 (assert (fk(column ?a00)(table ?c30)(fktable ?c00)))
 (assert (column(name ?a00)(table ?c30)(pk 0)))
)
```

Règle idéale:

```
(defrule Create−FK−1n
 (association1n (classfr ?c00) (classto ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (column (name ?o20)(table ?c00)(pk 1))
 (table (name ?c00)(alternname ?t21))
 =>
 (assert (fk (column ?o20) (table ?c10) (fktable ?c00)))
 (assert (column (name ?o20)(table ?c10)(pk 0)))
)
```

Voici un autre exemple de règle générée. Elle permet de transformer une association 1-N en une clé étrangère. On peut constater que dans ses conditions, elle vérifie s'il existe un attribut unique dans la classe $c00$ afin de l'utiliser comme clé étrangère. Dans la règle idéale, la vérification se fait sur le modèle cible, donc sur l'existence d'une clé primaire. La règle générée semble correcte dans un premier temps. Malheureusement, elle ne peut pas fonctionner lorsqu'une table reçoit un identifiant généré dû à l'absence d'attribut unique dans la classe associée.

Règle générée:

```
(defrule R_23348250
 (association11(classa ?c00)(classb ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (test (eq (count−query−results :allAssociation ?c10 ) 1))
 =>
 (assert (table(name ?c00)(alternname ?c10)))
)
```

Règle idéale:

```
(defrule Detect−Class−Assoc−11bis
 (association11 (classa ?c00)(classb ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (not (inheritance (class ?c20)(superclass ?c10)))
 (class(name ?c10))

 (test (eq (count−query−results allAssociation ?c10) 1))
 =>
 (assert (table (name ?c00)(alternname ?c10)))
)
```

Cette règle a été générée pour fusionner des classes qui n'étaient reliées que par des associations de type 1-1. Malheureusement, l'exemple passé en entrée ne possédait pas de classe qui était à la fois une super-classe et qui était reliée que par une seule association de type 1-1. Cette règle a donc été jugée correcte par notre programme alors que pour d'autres exemples, il aurait fallu rajouter la condition qui vérifie si la classe ne possède pas une ou plusieurs sous-classes.

Parfois, le programme peut également générer des règles qui n'ont à première vue aucun sens. Mais combinées avec d'autres, elles peuvent transformer correctement un modèle source en un modèle cible attendu. On peut remarquer que plus le modèle passé en entrée est complexe et demande un plan de transformation proche de l'idéal, moins il existe de plans de transformation différents pouvant transformer ce modèle correctement.

Règle générée:

```
(defrule R_23389541
 (attribute (name ?a00)(class ?c00)(unico 1))
 (class(name ?c00))

 (table (name ?c00)(alternname ?t21))

 (attribute (name ?a40)(class ?t21)(unico ?a42))
 (class(name ?t21))
 =>
 (assert (column (name ?a40)(table ?c00)(pk 0)))
)
```

Règle idéale:

```
(defrule Transfo−Merged−Attribute−Column
 (attribute (name ?a00)(class ?c00)(unico ?a02))
 (class(name ?c00))

 (table (name ?t20)(alternname ?c00))

 (attribute (name ?a40)(class ?t20)(unico 1))
 (class(name ?t20))
 =>
 (assert (column (name ?a00)(table ?t20)(pk 0)))
)
```

Pour finir, il est possible de générer des règles qui semblent différentes mais qui exécutent exactement la même tâche. On constate ici que les briques ont été générées dans un ordre différent et qu'elles possèdent donc un nommage de variables différent. Malgré cela, les liaisons ont été réalisées de la même manière, les deux règles de transformation sont donc identiques. Ceci prouve qu'il existe plusieurs solutions différentes pour un même plan de transformation.

5.4 Évaluation quantitative

Les programmes générés disposent de propriétés telles que le nombre de règles contenues dans le programme mais aussi une valeur de fonction objectif. Il est dès lors possible de les évaluer de manière quantitative. Premièrement, elle permettra de s'assurer que les résultats sont toujours plus ou moins identiques pour un même exemple et qu'il ne s'agit donc pas de "chance". Deuxièmement, elle permettra de s'assurer que les résultats sont meilleurs que ceux de la solution initiale.

Les exécutions mises en place pour l'évaluation étaient configurées avec une population de 200 à 400 programmes et avec un nombre de génération compris entre 2000 et 4000. Un cinquième de la population était considéré comme les programmes faisant partie de l'élite et les plans de transformation étaient générés avec un maximum de six modules. Pour ce genre de configuration, l'exécution pouvait prendre un à trois jours complets pour les exemples les plus complexes.

Dans un premier temps, le niveau de complexité des exemples étaient considérés de manière intuitive. Mais avec l'aide de la récolte des résultats, il a été constaté qu'il était possible d'évaluer la complexité d'un modèle selon le nombre de règles minimum nécessaire que le programme doit générer pour atteindre une fonction objectif de 100%. Plus le plan de transformation a besoin de règles pour transformer le modèle correctement, plus ce modèle possède de particularités.

Les plans de transformation nécessitant au maximum 5 à 6 règles pour résoudre la transformation de la paire d'exemples fournie en entrée ont pu être générés. Il a été possible de générer un plan de transformation nécessitant 11 règles. Mais, généralement, les résultats obtenus pour ceux nécessitant un plus grand nombre de règles atteignaient un score de 80-90%. L'espace de recherche étant vraiment trop important et le programme n'arrivait pas à converger vers le plan de transformation idéal.

5.4.1 Évaluations multiples sur un même exemple

Afin de vérifier que les bons résultats ne sont pas de simples coups de chance, plusieurs exécutions sur la même paire d'exemple ont été effectuées (fig. 5.2 et fig. 5.3).

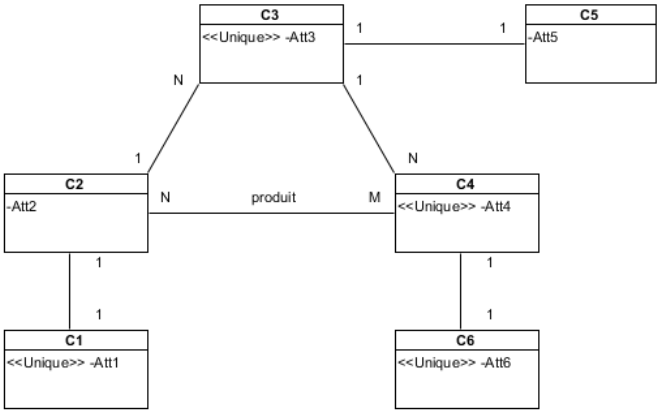

Figure 5.2: *Exemple de diagramme de classe*

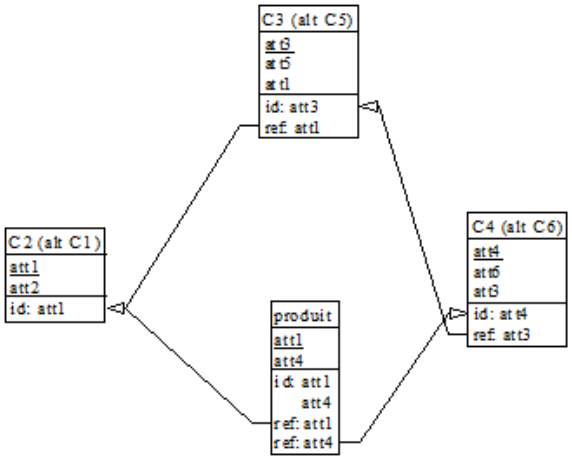

Figure 5.3: *Exemple de schéma relationnel*

Trois exécutions de 2000 générations de 200 programmes ont été lancées. Comme on peut le voir sur le graphique ci-dessous (fig. 5.4), les résultats obtenus sont assez disparates puisqu'on obtient 0.4588, 0.4828 et 0.5119, soit respectivement 76.4%, 80.4% et 85.3%. De plus, les courbes ne bougent pas énormément.

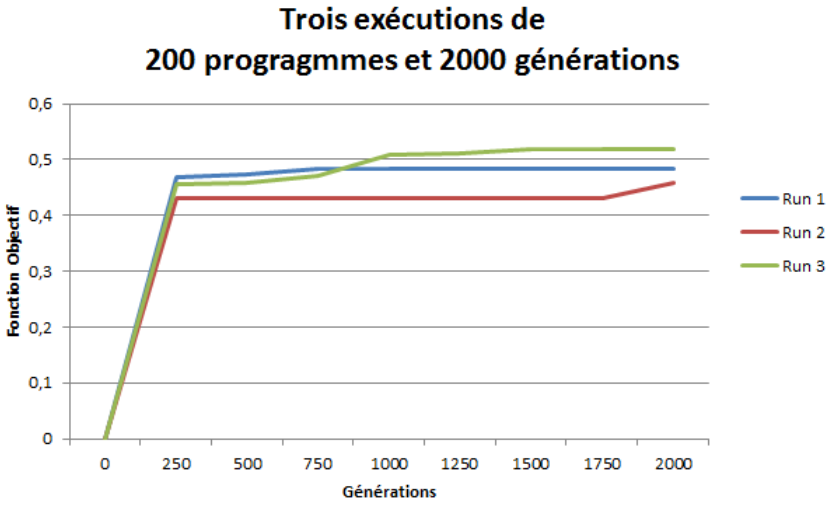

Figure 5.4: *Résultats de trois exécutions de 2000 générations de 200 programmes*

Trois autres exécutions ont été effectuées avec cette fois-ci 4000 générations de 400 programmes. Le graphe ci-dessous (fig. 5.5) montre que les exécutions ont tendance à converger vers la même valeur de 0.52, soit 86.6%. Les courbes ont tendance à bouger plus et cela s'explique sans doute par le fait que les populations sont plus grandes et ont donc plus de chance de donner de meilleurs programmes.

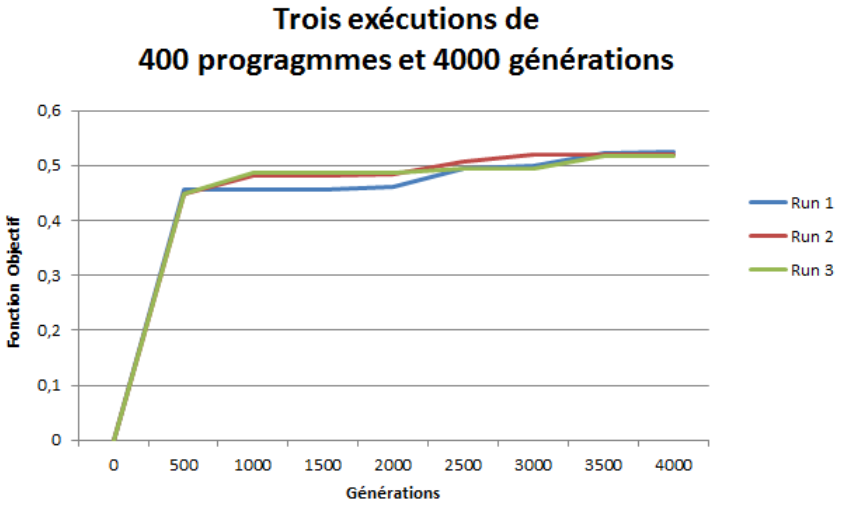

Figure 5.5: *Résultats de trois exécutions de 4000 générations de 400 programmes*

5.4.2 Évaluations sur plusieurs exemples

Afin de prouver que les résultats sont meilleurs qu'avec la solution initiale, plusieurs exécutions ont été effectuées sur des exemples différents. Toutes donnent d'assez bon résultats allant de 0.512 à 0.586, soit entre 85.3% et 97.6%, alors que l'approche initiale avait des scores oscillant entre 0.3 et 0.4 soit entre 50 et 60%. Le nombre de règle varie entre 5 et 10. Ci-dessous, quatre graphiques générés pour les quatre exemples (fig. 5.6 et fig. 5.7).

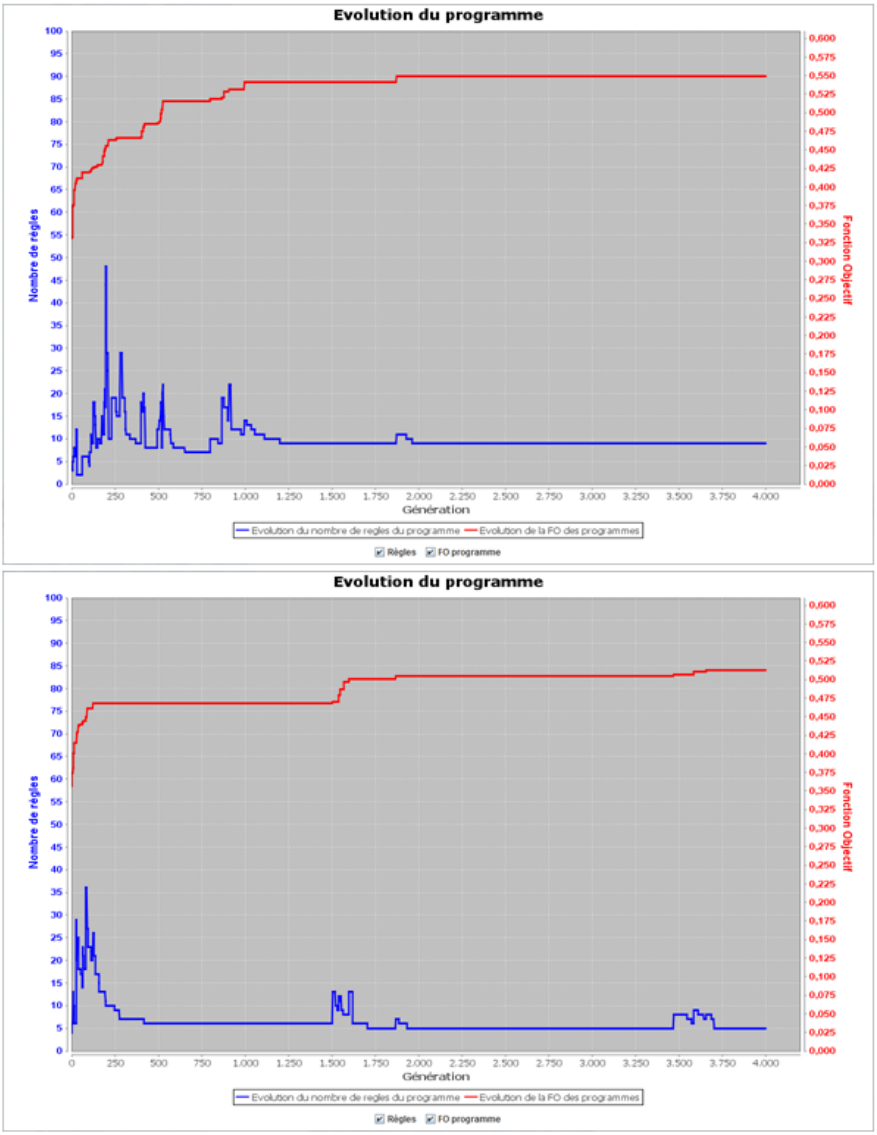

Figure 5.6: *Résultats des exemples 1 et 2*

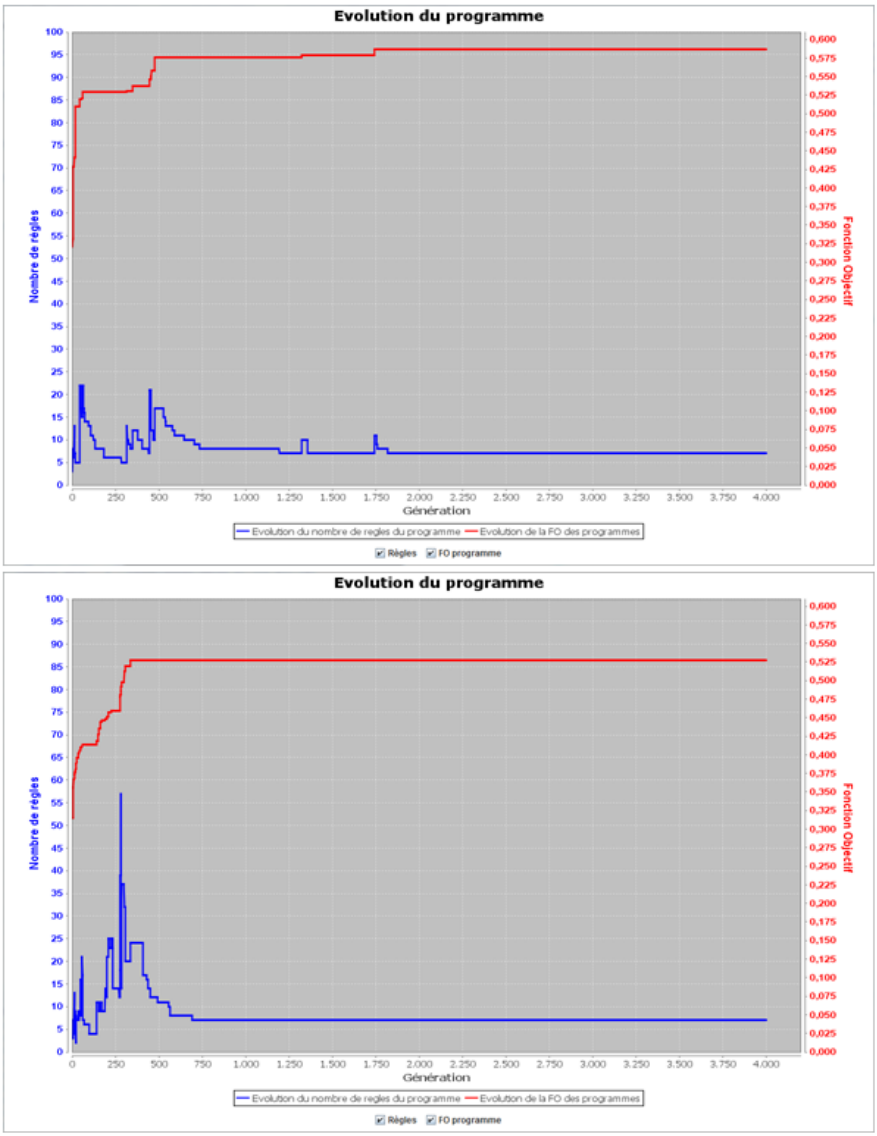

Figure 5.7: *Résultats des exemples 3 et 4*

Comme on peut le voir sur les différents graphes, les résultats sont assez proches du maximum. On remarque également qu'il y a beaucoup d'activité durant les mille premières générations. C'est évident, plus on avance, meilleurs deviennent les programmes. Il devient donc difficile d'améliorer la solution après avoir atteint un score de 0.5 de fitness.

5.5 Testeur de règles

Afin de s'assurer de la validité des mécanismes développés pour créer des règles et des mutations, un testeur de règles a été implémenté. L'idée consiste, d'une part, à générer des règles aléatoires mais respectant un certain nombre de conditions (nombre de briques de la partie condition, nombre de faits dans la partie action, nombre de *NOT*, nombre de conditions portant sur le modèle cible, etc.). Les règles générées sont alors comparées avec une chaîne de caractères représentant la règle idéale que l'on souhaite trouver. D'autre part, le testeur est capable de prouver que les mutations fonctionnent, en prenant une règle générée aléatoirement et en lui appliquant, de manière successive, une même mutation, jusqu'à tomber sur une règle désirée, fournie en paramètre du testeur. Ce testeur permet donc de repérer les règles non-générables mais également la difficulté à générer certaines règles du programme idéal.

Bien que toutes les règles du programme idéal soient générables, certaines sont plus difficiles à générer que d'autres. La règle chargée de générer des identifiants techniques demande en moyenne 10.000 itérations avant de tomber sur la règle souhaitée, tout comme la règle traitant de l'héritage. Les règles chargées de traiter les clés étrangères pour les relation N-M (avec ou sans classe d'association) sont encore plus difficiles à générer puisqu'il faut pas loin de 30.000 itérations.

Bien évidemment, plus le squelette d'une règle est important, plus le nombre d'itérations nécessaires pour tomber dessus augmentera. Il faut cependant regarder ces résultats avec un certain recul. En effet, les règles idéales fournies en entrée sont fortement figées puisqu'elles contiennent des noms de variables arbitraires (mais générables). Pourtant, une autre configuration, avec des noms de variable différents, marcherait. La chance intervient aussi dans ce processus purement aléatoire et il n'est pas exclu que l'on tombe sur la bonne règle en une seule itération au lieu de 30.000. Néanmoins, ces chiffres reflètent la difficulté à générer certaines "grosses" règles.

Le testeur permet simplement de s'assurer que toutes les règles du programme idéal sont générables par l'approche RAC ou RTC et que les mutations sont fonctionnelles.

5.6 Discussion

Outrepasser les limites de l'approche initiale, en gérant, par exemple, les associations 1-1 et la génération d'identifiants techniques, a eu pour effet d'augmenter le nombre de règles nécessaire à la transformation de diagramme de classe en schéma relationnel (20 règles dans le programme idéal) et les a fortement complexifié (plusieurs blocs dans les parties gauches et droites des règles). Cela a eu un impact direct sur l'agrandissement de l'espace de recherche, rendant la convergence plus longue et difficile.

Sur des exemples de petites tailles et relativement simples, l'approche initiale pourrait donc être plus performante étant donné son espace de recherche plus réduit. Mais rien n'empêche l'approche RAC d'aller plus vite avec un peu de chance. En revanche, l'approche initiale obtient de faibles résultats lorsque les exemples se complexifient alors que l'approche RAC, bien qu'elle n'atteigne pas des scores de 100%, donne de bons résultats, proches du maximum.

Un des problèmes aperçu lors de l'évaluation quantitative était la difficulté pour l'application de converger vers la solution. Une étude heuristique pour aider l'algorithme à converger n'était pas le but de ce travail. Par contre, il fallait garantir que chaque règle de notre programme idéal pouvait être générée par l'application. Le testeur de règle a permis de s'en assurer.

Un autre problème aperçu plus tôt pendant la vérification qualitative des résultats était que certains programmes pouvaient être corrects mais seulement pour une paire d'exemples spécifiques. La solution trouvée marchait pour un exemple en particulier, mais donnait de moins bons résultats sur une autre paire d'exemples. Par contre, plus l'exemple était complexe, plus le plan de transformation pouvait être généralisé à d'autres exemples. Une solution proposée dans l'application de départ, mais qui n'a pas encore été implémentée dans cette version, est de pouvoir évaluer les programmes générés non pas sur une paire d'exemples, mais bien sur un ensemble fini de pair d'exemples.

L'approche RTC n'ayant pas été assez testée, la base de résultats n'est pas assez complète pour pouvoir proposer une analyse qualitative et quantitative. Néanmoins, les résultats semblaient légèrement moins bons.

Chapitre 6

Conclusion

6.1 Résumé de la contribution

Le but de ce mémoire était de poursuivre le développement d'une approche de génération de transformation de modèle par l'exemple. Après analyse de cette approche existante, ses limites ont pu être identifiées et notre contribution a permis de les atténuer ou de les faire disparaître. L'approche initiale était fonctionnelle mais limitée à des cas de transformation de modèles extrêmement simplifiés. Notre intervention permet d'exploiter des métamodèles plus complets et plus proches de ce qui est utilisé en général dans la pratique.

L'approche présentée dans ce mémoire est donc une approche de type M2M. Même si les modèles sont exprimés au format textuel (en JESS) pour les besoins de l'application, ils sont la traduction de modèles graphiques, tels que ceux proposés par UML.

Étant donné que l'on utilise JESS, l'approche est de type déclarative, puisque l'on utilise des prédicats de type Prolog. Mais la dimension de contrôle, l'un des problèmes clés de ce travail, en fait aussi une approche impérative. De l'approche initiale purement déclarative, nous sommes passés à une approche hybride, mêlant, à la fois, des règles écrites en un langage proche de Prolog et un mécanisme de modules permettant d'ordonnancer le déclenchement des règles. Mais il ne s'agit pas d'une approche hybride au sens de la définition donnée dans le deuxième chapitre. L'utilisateur ne choisit pas s'il travaille de manière déclarative ou impérative comme avec QVT par exemple. Il travaille des deux façons simultanément.

On peut également analyser l'approche présentée dans ce mémoire selon le feature diagram de Czarnecki et Helsen (cf. section 2.3.3):

- Les règles sont composées d'une partie gauche, relative aux conditions, et d'une partie droite, relative aux actions. Contrairement à la définition donnée par Czarnecki et Helsen, le côté gauche des règles de l'approche présentée dans ce mémoire n'est pas restreint au modèle source, mais peut également contenir des conditions portant sur le modèle cible. Le côté droit est, lui, bien restreint au modèle cible.

- Il n'est malheureusement pas possible de spécifier de périmètre d'application des règles dans les modèles sources et cibles. L'approche transforme les modèles de façon intégrale.

- L'approche permet d'opérer des transformations endogènes ou exogènes, et horizontales ou verticales. Mais celle-ci a essentiellement été testée sur des transformations de type exogène-horizontale.

- La stratégie d'application des règles est, dans cette approche, de type non-déterministe.

- Pour la planification des règles, l'approche permet de spécifier un ordre à la fois implicite et explicite d'exécution des règles. Les règles peuvent être regroupées explicitement dans des structures (les modules). Le moteur d'inférence choisira lui-même l'ordre d'application des règles au sein de ce module. L'approche peut également fonctionner de manière totalement explicite en définissant un numéro d'ordre des règles (salience).

- La traçabilité n'est pas utilisée ni implémentée dans les deux approches présentées dans ce mémoire.

- L'approche n'est conçue que pour évaluer des règles unidirectionnelles.

Au contraire de la majorité des travaux présentés dans le deuxième chapitre, les approches RAC et RTC soulagent l'utilisateur de la conception des règles de transformation. De plus, les règles créées sont plus puissantes dans la mesure où il est possible de tester le méta-modèle cible, de créer des conditions complexes, de tester la non-existence de faits, de produire plus d'une construction dans le modèle cible, ou encore d'utiliser des primitives de navigations, choses que peu d'approches permettent de réaliser.

L'un des besoins des langages de transformation de modèle est la *réutilisation* (cf. section 2.3.5). Pouvoir conserver le plan de transformation généré et l'appliquer à d'autres modèles compatibles est l'intérêt premier de ces approches. L'aspect *configuration* (un autre besoin) de ce travail

est également l'un de ses points forts. Il est en effet possible de configurer l'application pour transformer n'importe quel modèle en un autre, en lui spécifiant les deux méta-modèles à utiliser et d'autres propriétés via un fichier de configuration relativement complet.

Avec les deux approches proposées dans ce mémoire, nous apportons donc notre pierre à l'édifice de la résolution du problème de la transformation de modèles par l'exemple. Les résultats présentés dans le chapitre précédent prouvent que la transformation de modèles par l'exemple est une approche viable et efficace. Couplée à la programmation génétique, cette technique est idéale pour atteindre un haut degré d'automatisation, qui sera bénéfique au maintient de la cohérence entre les modèles utilisés au sein du développement de systèmes informatiques.

6.2 Résumé des résultats

Le chapitre 4 nous a permis de voir que l'approche RAC était largement plus puissante que l'approche initiale, dans le sens où elle permet de gérer des transformations de diagrammes de classe vers des schémas relationnels qui sont plus complexes et plus proches de ce que l'on pourrait trouver dans la réalité, même si l'écart est encore important en termes de taille des modèles et de complétude des méta-modèles utilisés.

Là où l'approche initiale peinait à dépasser des scores de 65%, l'approche RAC arrivait, en général, à des scores allant de 85 à 100%, ce qui constitue donc une belle progression.

Aussi, bien que nous ne soyons pas parvenus à obtenir une copie conforme du programme de transformation idéal, un testeur de règles a permis de montrer que cela était techniquement faisable. Cependant, le facteur chance joue énormément étant donné le caractère (fortement) aléatoire des opérateurs génétiques.

6.3 Limitations

Bien que la solution initiale fut limitée dans ses résultats, elle pouvait néanmoins être utilisée pour traiter trois types de transformation de modèles:

- Diagramme de classe vers schéma relationnel
- Diagramme de séquence vers machines à états
- Diagramme de séquence complexe vers machines à états

Par manque de temps, les améliorations que nous avons pu apporter à cette approche n'ont été testées que sur les transformations de type "diagramme de classe vers schéma relationnel". Cependant, l'approche a pour but de traiter tous les types de transformation possibles, et ce même avec des méta-modèles encore inconnus à ce jour. Il y a donc de fortes chances pour que les modifications apportées soient tout autant bénéfiques pour les deux autres types de transformation mentionnés ci-dessus, mais nous ne pouvons l'affirmer avec certitude.

L'une des particularités de l'approche de transformation que nous avons abordé est le fait qu'elle utilise des exemples pour générer des règles de transformation. Cependant, bien que les exemples utilisés soient corrects, aussi bien sur le plan syntaxique que sémantique, ils ne couvrent pas l'ensemble des possibilités offertes par leur méta-modèle respectif. Il est, dès lors, compliqué de générer un programme de transformation de modèles qui serait adapté à l'entièreté des modèles pouvant être créés à partir des mêmes méta-modèles. Bien que certains exemples aient donné des résultats de 100%, indiquant que la transformation générée permet d'arriver au résultat escompté (sans surplus), cela ne signifie pas que la solution trouvée est la meilleure solution. Les plans de transformation générés étaient corrects mais spécifiques à un exemple en particulier, et rien ne garantit que l'application de ce plan de transformation sur une autre paire d'exemple donnera un score de 100% également.

Malheureusement, étant donné les importantes modifications apportées à la solution initiale, l'espace de recherche pour trouver des plans de transformation a fortement augmenté. Le nombre de générations nécessaire à la dérivation d'un plan de transformation a dû être revu à la hausse, rendant la recherche plus longue, plus difficile et plus gourmande en temps CPU et en mémoire (multiplication par un facteur 4). Une exécution complète, à savoir 4000 générations de 400 programmes, est très longue, et ce même sur un serveur doté de 96Gb RAM et de plusieurs processeur Intel XEON. Le temps nécessaire pour trouver une solution avoisine aujourd'hui les 72h. Néanmoins, les performances des ordinateurs évoluant sans cesse, le temps pour trouver une solution devrait devenir raisonnable dans quelques années.

Il est cependant important de signaler que le programme de transformation "idéal" pour traiter les transformations de type "diagramme de classe vers schéma relationnel" comporte actuellement vingt règles. Certaines d'entre-elles étant relativement complexes, cela peut expliquer un temps de convergence relativement long. Il est possible que d'autre types de transformation s'accommodent d'un nombre de règles bien moindre et que celles-ci soient moins complexes à générer. La difficulté résidera donc dans un paramétrage efficace de l'application afin d'"aider" le programme à

converger rapidement.

Aussi, bien que nous puissions traiter des exemples plus complexes que l'approche initiale, les méta-modèles utilisés ne sont pas des méta-modèles officiels et complets. On regrettera l'absence de relation de compositions et d'agrégations dans les diagrammes de classe par exemple.

Il est nécessaire de prédéfinir les briques pour chaque type de modèle qu'on souhaite transformer. Le travail n'ayant été évalué que sur les diagrammes de classes et les schémas relationnels, la numérotation pour identifier de façon précise une propriété a été implémentée naïvement. Par exemple, en regardant la règle ci-dessous, on peut remarquer que la valeur de la propriété *name* du premier constructeur *class* a été nommée *c*00. Le premier 0 correspond à l'identifiant du constructeur et le deuxième 0 correspond à l'identifiant du slot dans ce constructeur. Étant donné qu'il n'existe pas de brique demandant plus de deux identifiants de constructeur dans les méta-modèles de diagramme de classe et de schéma relationnel, la numérotation est simplement incrémentée par 2 entre chaque brique.

```
(defrule Inheritance
 (inheritance (class ?c00)(superclass ?c10))
 (class(name ?c00))
 (class(name ?c10))

 (column (name ?o20)(table ?c10)(pk 1))
 (table (name ?c10)(alternname ?t21))
 =>
 (assert (fk(column ?o20)(table ?c00)(fktable ?c10)))
 (assert (column(name ?o20)(table ?c00)(pk 1)))
)
```

Il serait plus judicieux d'implémenter des identifiants spécifiques, propres aux concepts de brique, constructeur et propriété (slot).

Un problème qui pourrait se poser dans le futur est le fait que les transformations générées sont étroitement liées aux méta-modèles desquels elles sont issues. Cela signifie que si un méta-modèle venait à être modifié pour une quelconque raison, la transformation dérivée à partir de la précédente version ne serait plus valable, et une nouvelle exécution serait alors nécessaire afin de dériver un nouveau plan de transformation capable de transformer des modèles issus de ce nouveau méta-modèle. Certaines approches, comme ATL par exemple, proposent une "co-évolution" automatique entre les méta-modèles et les règles de transformation. Les règles de transformation évoluent en parallèle avec les méta-modèles, évitant ainsi de devoir réécrire les règles à chaque nouvelle version de méta-modèle. On ne

rencontre donc pas la caractéristique désirable de la maintenance (cf. section 2.3.5).

Une autre caractéristique désirable concerne le fait que les transformations soient claires, concises et précises. Malheureusement, étant donné que, dans cette approche, les règles sont générées aléatoirement, celles-ci ne sont pas aussi bien écrites que celles du programme de transformation idéal. La composition de transformation n'est malheureusement pas non plus possible.

Enfin, l'approche RTC (Rules Then Control) a été développée assez tardivement et n'a pas pu être validée complètement. Bien que l'idée semble marcher, il est impossible d'affirmer qu'elle est plus ou moins efficace que l'approche RAC (Rules And Control) pour le moment.

6.4 Travaux futurs

Étant donné que nous nous situons dans un cadre de recherche scientifique, il est évident que l'approche qui a été développée fera l'objet de travaux futurs. Ceux-ci porteront essentiellement sur la réponse aux limitations identifiées dans la section précédente. Plusieurs pistes peuvent d'ores et déjà être proposées:

1. Concernant le problème des exemples fournis en entrée, jugé insuffisamment expressifs, deux solutions peuvent être proposées:

   - Utiliser un exemple reprenant l'entièreté des possibilités offertes par le méta-modèle, ce qui donnerait lieu à un exemple de taille relativement importante. Cependant cela ne sera pas trop pénalisant du point de vue calculs à effectuer. En effet, il ne s'agit que de faits à traiter et bien qu'il y en aura sans doute beaucoup, l'opération restera rapide. L'inconvénient est qu'il faudra réaliser une paire d'exemples relativement complexe. Or l'avantage des approches "par l'exemple" est de permettre à l'utilisateur de rester à l'écart de la complexité intrinsèque aux méta-modèles.
   - Utiliser une base d'exemples, c'est-à-dire un ensemble d'exemples de tailles et complexités variées mais qui, ensemble, couvrent les différents aspects du méta-modèle. Les programmes de transformation de modèles devront alors être successivement testés sur les différents exemples de la base. Ce processus sera, lui, coûteux en temps car s'il l'on fournit 10 exemples, chaque programme devra être évalué 10 fois... Le processus sera donc 10 fois plus long que le processus actuel. L'avantage est que l'utilisateur n'a pas à fournir des exemples trop compliqués mais suffisamment variés et expressifs.

2. L'idée d'utiliser le recuit simulé dans l'approche RAC semble intéressante, pour renforcer la dimension de contrôle. Cependant, le processus durerait encore plus longtemps.

3. Bien sûr, la prise en charge des autres méta-modèles constitue une amélioration et la définition des "briques" propres à ces méta-modèles devra être effectuée.

4. Une révision des méta-modèles utilisés peut être envisagée, afin de fournir des moyens plus efficaces et rapides pour générer les règles de transformation. On pensera notamment aux types d'associations valables dans un seul sens, impliquant la génération d'une double règle pour prendre en charge les deux sens du lien. JESS propose par exemple l'utilisation de multislots qui peuvent être instanciés avec un ensemble de valeurs contrairement aux slots ne pouvant recevoir qu'une valeur unique. Ces multislots pourraient par exemple simplifier et réduire le nombre de règles impliquées pour la gestion des associations.

5. Malheureusement, même si cette optimisation pourrait être bénéfique pour des diagrammes de type structurel comme les diagrammes de classe, d'autres diagrammes, plutôt de type dynamique, pourraient avoir besoin de générer des règles de transformation où le sens d'une association est important. En gérant l'ordonnancement de ces multislots, le sens pourrait alors être défini.

6. L'optimisation de l'espace de recherche est importante et devra faire l'objet de recherche pour aider l'approche à converger plus rapidement.

7. L'ajout de nouveaux opérateurs sera peut-être nécessaire pour prendre en charge d'autres méta-modèles, mais ces opérateurs agrandissent souvent l'espace de recherche de manière très importante. L'opérateur OR a par exemple été écarté car la logique du premier ordre nous garantit que les opérateurs AND et NOT permettent de générer toutes les possibilités.

8. Il sera également question de rendre les exemples les plus complets possible en prenant en charge toutes les possibilités des méta-modèles réels, telles que les relations de compositions/agrégation dans le diagramme de classe.

Bibliographie

[1] *Codagen architect 3.2.* http://www.omg.org/mda/mda_files/Codagen2004.pdf.

[2] *Graphical modeling framework.* http://www.eclipse.org/modeling/gmp/.

[3] *Metaedit+.* http://www.metacase.com/products.html.

[4] *Optimalj 4.3.* http://www.componentsource.com/beaws/products/compuware-optimalj-professional-eclipse-named-users/index.html.

[5] *Rational xde.* http://www.ibm.com/developerworks/rational/products/xde/.

[6] *Xslt.* http://www.w3.org/TR/xslt.

[7] *Xtext.* http://www.eclipse.org/Xtext/.

[8] *Andromda 2.0.3.* http://www.andromda.org, 2003.

[9] *Arcstyler 4.0.* http://cc.codegear.com/partners/jbuilderx/interactive_objects_software/arcstyler_4_0/index.html, 2004.

[10] *Jamda: The java model driven architecture 0.2.* http://sourceforge.net/projects/jamda, 2005.

[11] D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier, *Sitra: simple transformations in java*, in Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems, MoDELS'06, Berlin, Heidelberg, 2006, Springer-Verlag, pp. 351–364.

[12] D. H. Akehurst and S. Kent, *A relational approach to defining transformations in a metamodel*, in Proceedings of the 5th International Conference on The Unified Modeling Language, UML '02, London, UK, UK, 2002, Springer-Verlag, pp. 243–258.

[13] M. Alanen, I. Porres, T. Centre, and C. Science, *I.: Difference and union of models*, in UML 2003. LNCS, Springer, 2003, pp. 2–17.

[14] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, *Uml2alloy: A challenging model transformation*, 2007, pp. 436–450.

[15] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai, *The graph rewriting and transformation language: Great*, Electronic Communications of the EASST, 1 (2006).

[16] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu, *Barriers to systematic model transformation testing*, Commun. ACM, 53 (2010), pp. 139–143.

[17] A. Belbachir, R. Deau, R. Lenne, and J. Snoussi, *La programmation génétique.* `http://liris.cnrs.fr/alain.mille/enseignements/master_ia/rapports_2006/Programmation%20Genetique_4p.pdf`, 2006.

[18] J. Bézivin and O. Gerbé, *Towards a precise definition of the omg/mda framework*, in Proceedings of the 16th IEEE international conference on Automated software engineering, ASE '01, Washington, DC, USA, 2001, IEEE Computer Society, pp. 273–280.

[19] b+m Informatik AG, *open architectureware: Generator framework.* `http://architecturware.sourceforge.net`, 2005.

[20] P. Braun, F. Marschall, A. R. Vorbehalten, P. Braun, F. Marschall, and T. U. München, *Botl – the bidirectional object oriented transformation language*, tech. report, 2003.

[21] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, *Stratego/xt 0.16: components for transformation systems*, in Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM '06, New York, NY, USA, 2006, ACM, pp. 95–99.

[22] P. Brosch, M. Seidl, K. Wieland, M. Wimmer, and P. Langer, *The operation recorder: specifying model refactorings by-example*, in Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, OOPSLA '09, New York, NY, USA, 2009, ACM, pp. 791–792.

[23] J. Cabot, R. Clarisó, and D. Riera, *Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming*, in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ASE '07, New York, NY, USA, 2007, ACM, pp. 547–548.

[24] CBOP, DSTC, and IBM, *Mof query/views/transformations, revised submission. omg document: ad/03-08-03*, 2003.

[25] J. R. Cordy, *The txl source transformation language*, Sci. Comput. Program., 61 (2006), pp. 190–210.

[26] A. L. Correa and C. M. L. Werner, *Applying refactoring techniques to uml/ocl models.*, in UML, T. Baar, A. Strohmeier, A. M. D. Moreira, and S. J. Mellor, eds., vol. 3273 of Lecture Notes in Computer Science, Springer, 2004, pp. 173–187.

[27] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró, *Viatra - visual automated transformations for formal verification and validation of uml models*, in Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on, 2002, pp. 267–270.

[28] K. Czarnecki and S. Helsen, *Classification of model transformation approaches*, 2003.

[29] J. M. Daida, J. D. Hommes, T. F. Bersano-Begey, S. J. Ross, and J. F. Vesecky, *Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice*, in Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr., eds., MIT Press, Cambridge, MA, USA, 1996, ch. 21, pp. 417–442.

[30] C. Darwin, *On the origin of species*, New York : D. Appleton and Co.,, 1859. http://www.biodiversitylibrary.org/bibliography/28875.

[31] J. de Lara and H. Vangheluwe, *Atom3: A tool for multi-formalism and meta-modelling.*, in FASE, R.-D. Kutsche and H. Weber, eds., vol. 2306 of Lecture Notes in Computer Science, Springer, 2002, pp. 174–188.

[32] V. Englebert, J. Henrard, J.-M. Hick, D. Roland, and J.-L. Hainaut, *Db-main: un atelier d'ingénierie de bases de données.*, in BDA, G. Jomier, ed., INRIA, 1995, pp. 345–364.

[33] M. Faunes, H. A. Sahraoui, and M. Boukadoum, *Genetic-programming approach to learn model transformation rules from examples*, in ICMT, 2013, pp. 17–32.

[34] C. Fonlupt, *La programmation génétique*. `http://www-lil.univ-littoral.fr/~fonlupt/Recherche/PG/prog-genetique.html`, 2010.

[35] A. A. Freitas, *Data Mining and Knowledge Discovery with Evolutionary Algorithms*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[36] I. García-Magariño, J. Gómez-Sanz, and R. Fuentes-Fernández, *Model transformation by-example: An algorithm for generating many-to-many transformation rules in several model transformation languages*, in Theory and Practice of Model Transformations, R. Paige, ed., vol. 5563 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 52–66.

[37] J. J. C. Gomez, B. Baudry, and H. Sahraoui, *Searching the boundaries of a modeling space to test metamodels*, in Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, Washington, DC, USA, 2012, IEEE Computer Society, pp. 131–140.

[38] N. Habra, *Ingénierie du logiciel*. University Lecture, 2012.

[39] J.-L. Hainaut, *Bases de données : Concepts, utilisation et développement*, 2009, pp. 487–490.

[40] S. Handley, *Classifying nucleic acid sub-sequences as introns or exons using genetic programming*, in Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology, Cambridge, United Kingdom, July 16-19, 1995, C. J. Rawlings, D. A. Clark, R. B. Altman, L. Hunter, T. Lengauer, and S. J. Wodak, eds., AAAI, 1995, pp. 162–169.

[41] S. Hidaka, Z. Hu, H. Kato, and K. Nakano, *Towards a compositional approach to model transformation for software development*, in Proceedings of the 2009 ACM symposium on Applied Computing, SAC '09, New York, NY, USA, 2009, ACM, pp. 468–475.

[42] E. J. F. Hill, *Jess, the Java Expert System Shell*, tech. report, SANDIA National Laboratories, 2000.

[43] IKV++ Technologies, *mediniQVT*.

[44] D. Jackson, *Alloy: a lightweight object modelling notation*, ACM Trans. Softw. Eng. Methodol., 11 (2002), pp. 256–290.

[45] J.-M. Jézéquel, O. Barais, and F. Fleurey, *Model driven language engineering with kermeta*, in Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III, GTTSE'09, Berlin, Heidelberg, 2011, Springer-Verlag, pp. 201–221.

[46] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, *Atl: A model transformation tool*, Sci. Comput. Program., 72 (2008), pp. 31–39.

[47] F. Jouault, J. Bézivin, and I. Kurtev, *Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering*, in Proceedings of the 5th international conference on Generative programming and component engineering, GPCE '06, New York, NY, USA, 2006, ACM, pp. 249–254.

[48] R. Jubeh, A. Koch, and A. Zündorf, *Uml1.4 to 2.1 activity diagram model migration with fujaba - a case study*, in Transformation Tool Contest 2010, S. Mazanek, A. Rensink, and P. van Gorp, eds., 2010, pp. 54 – 60.

[49] A. Kalnins, J. Barzdins, and E. Celms, *Model transformation language mola*, in in: Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004, 2004, pp. 14–28.

[50] M. Kessentini, H. Sahraoui, and M. Boukadoum, *Model transformation as an optimization problem.*, Berlin: Springer, 2008, pp. 159–173.

[51] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, Science, 220 (1983), pp. 671–680.

[52] A. Königs, *Model transformation with triple graph grammars*, in Model Transformations in practice Satellite workshop of Models 2005, MONTEGO, 2005.

[53] D. S. Kolovos, R. F. Paige, and F. A. Polack, *The epsilon transformation language*, in Proceedings of the 1st international conference on Theory and Practice of Model Transformations, ICMT '08, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 46–60.

[54] J. R. Koza, *Concept formation and decision tree induction using the genetic programming paradigm.*

[55] J. R. Koza, *Genetic programming.* `http://www.genetic-programming.com`.

[56] J. R. Koza, *Genetic programming - on the programming of computers by means of natural selection*, Complex adaptive systems, MIT Press, 1993.

[57] J. R. Koza, F. H. B. III, and A. David, *Using programmatic motifs and genetic programming to classify protein sequences as to cellular location*, 1998.

[58] J. R. Koza, M. A. Keane, M. J. Streeter, T. P. Adams, and L. W. Jones, *Invention and creativity in automated design by means of genetic programming*, AI EDAM: Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 18 (2004), pp. 245–269.

[59] J. M. Küster, *Definition and validation of model transformations*, Software and Systems Modeling, V5 (2006), pp. 233–259.

[60] P. Langer, M. Wimmer, and G. Kappel, *Model-to-model transformations by demonstration*, in Proceedings of the Third international conference on Theory and practice of model transformations, ICMT'10, Berlin, Heidelberg, 2010, Springer-Verlag, pp. 153–167.

[61] M. Lawley and J. Steel, *Practical declarative model transformation with tefkat*, in Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, J.-M. Bruel, ed., vol. 3844 of Lecture Notes in Computer Science, Springer, 2005, pp. 139–150.

[62] Y.-S. Lee and L.-I. Tong, *Forecasting time series using a methodology based on autoregressive integrated moving average and genetic programming*, Know.-Based Syst., 24 (2011), pp. 66–72.

[63] M2T, *Jet (java emitter templates)*, http://www.eclipse.org/modeling/m2t/?project=jet#jet.

[64] S. Markovic and T. Baar, *Refactoring OCL annotated UML class diagrams*, vol. 3713, 2005, pp. 280–294.

[65] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[66] T. Mens, *Model transformation: A survey of the state of the art*, tech. report, 2013.

[67] T. Mens, K. Czarnecki, and P. V. Gorp, *A taxonomy of model transformation*, in Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales

Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Electronic, 2005.

[68] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, *Equation of State Calculations by Fast Computing Machines*, The Journal of Chemical Physics, 21 (1953), pp. 1087–1092.

[69] MOF, *Meta object facility (mof) 2.0 core specification*, Tech. Report formal/06-01-01, OMG, 2001. OMG Available Specification.

[70] G. C. Murphy, D. Notkin, and K. J. Sullivan, *Software reflexion models: Bridging the gap between design and implementation*, IEEE Trans. Softw. Eng., 27 (2001), pp. 364–380.

[71] A. Narayanan and G. Karsai, *Towards verifying model transformations*, Electronic Notes in Theoretical Computer Science, 211 (2008), pp. 191–200.

[72] OMG, *Uml. unified modeling language specification.* `http://technav.ieee.org/tag/1933/unified-modeling-language`, 2003.

[73] OMG, *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[74] OMG, *Object Constraint Language Object Constraint Language, OMG Available Specification, Version 2.0*, 2006.

[75] OMG, *XML Metadata Interchange*, 2007.

[76] M. Peltier, J. Bézivin, and G. Guillaume, *Mtrans: A general framework, based on xslt, for model transformations.*

[77] J. L. Peterson, *Petri nets*, ACM Comput. Surv., 9 (1977), pp. 223–252.

[78] D. Pollet, *Une architecture pour les transformations de modèles et la restructuration de modèles uml*, these, Université Rennes 1, June 2005.

[79] I. Porres, *Model refactorings as rule-based update transformations.*, in UML, P. Stevens, J. Whittle, and G. Booch, eds., vol. 2863 of Lecture Notes in Computer Science, Springer, 2003, pp. 159–174.

[80] QVT-Partners, *Revised submission for mof 2.0 query / views / transformations rfp*, 2003.

[81] C. Raistrick, P. Francis, and J. Wright, *Model Driven Architecture with Executable UML(TM)*, Cambridge University Press, New York, NY, USA, 2004.

[82] C. R. Reeves, ed., *Modern heuristic techniques for combinatorial problems*, John Wiley & Sons, Inc., New York, NY, USA, 1993.

[83] J. Rothenberg, *The nature of modeling / Jeff Rothenberg*, The Rand Corporation Santa Monica, 1989.

[84] H. Saada, X. Dolques, M. Huchard, C. Nebut, and H. Sahraoui, *Generation of operational transformation rules from examples of model transformations*, in Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems, MODELS'12, Berlin, Heidelberg, 2012, Springer-Verlag, pp. 546–561.

[85] E. Seidewitz, *What models mean*, IEEE Softw., 20 (2003), pp. 26–32.

[86] S. Sendall and W. Kozaczynski, *Model transformation: the heart and soul of modeldriven software development*, Online]. Available: http://cui.unige.ch/ sendall/files/sendall-tech-report-EPFL-model-trans.pdf, pp. 42–45.

[87] Z. Somogyi, F. J. Henderson, and T. C. Conway, *Mercury, an efficient purely declarative logic programming language*, in In Proceedings of the Australian Computer Science Conference, 1995, pp. 499–512.

[88] J. Steel and M. Lawley, *Model-based test driven development of the tefkat model-transformation engine*, in Proceedings of the 15th International Symposium on Software Reliability Engineering, ISSRE '04, Washington, DC, USA, 2004, IEEE Computer Society, pp. 151–160.

[89] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, Addison-Wesley Professional, 2nd ed., 2009.

[90] H. Störrle and J. H. Hausmann, *Towards a Formal Semantics of UML 2.0 Activities*, 2005.

[91] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers, *Using description logic to maintain consistency between uml models.*, in UML, P. Stevens, J. Whittle, and G. Booch, eds., vol. 2863 of Lecture Notes in Computer Science, Springer, 2003, pp. 326–340.

[92] M. Strommer and M. Wimmer, *A framework for model transformation by-example: Concepts and tool support*, in Objects, Components, Models and Patterns, R. Paige and B. Meyer, eds., vol. 11 of Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 2008, pp. 372–391.

[93] Y. Sun, *Model transformation by demonstration*, in Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, OOPSLA '09, New York, NY, USA, 2009, ACM, pp. 831–832.

[94] I. O. P. Technology, *Revised submission for mof 2.0 query / views / transformations rfp*, 2003.

[95] L. Tratt, *Model transformations and tool integration*, 2004.

[96] M. van Kempen, M. Chaudron, D. Kourie, and A. Boake, *Towards proving preservation of behaviour of refactoring of uml models*, in Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, SAICSIT '05, Republic of South Africa, 2005, South African Institute for Computer Scientists and Information Technologists, pp. 252–259.

[97] B. Vanderose and N. Habra, *Tool-support for a model-centric quality assessment: Quatalog*, in Proceedings of the 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, IWSM-MENSURA '11, Washington, DC, USA, 2011, IEEE Computer Society, pp. 263–268.

[98] D. Varró, *Automated formal verification of visual modeling languages by model checking*, Software and Systems Modeling, V3 (2004), pp. 85–113.

[99] D. Varró, *Model transformation by example*, in Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems, MoDELS'06, Berlin, Heidelberg, 2006, Springer-Verlag, pp. 410–424.

[100] D. Varró and Z. Balogh, *Automating model transformation by example using inductive logic programming*, in Proceedings of the 2007 ACM Symposium on Applied Computing (SAC 2007), Seoul, Korea, March 11-15, 2007, Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo, eds., ACM Press, 2007, pp. 978–984.

[101] V. Černý, *Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm*, Journal of Optimization Theory and Applications, 45 (1985), pp. 41–51.

[102] G. Wachsmuth, *Metamodel adaptation and model co-adaptation*, in Proceedings of the 21st European conference on Object-Oriented Programming, ECOOP'07, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 600–624.

[103] E. D. Willink, *Umlx: A graphical transformation language for mda.*

[104] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler, *Towards model transformation generation by-example*, in Proceedings of the 40th Annual Hawaii International Conference on System Sciences, HICSS '07, Washington, DC, USA, 2007, IEEE Computer Society, pp. 285b–.

[105] J. Zhang, Y. Lin, and J. Gray, *Generic and domainspecific model refactoring using a model transformation engine," model-driven software development.*

[106] M. Zhang, U. Bhowan, and B. Ny, *Genetic programming for object detection: A two-phase approach with an improved fitness function*, Electronic Letters on Computer Vision and Image Analysis, 6 (2006), pp. 27–43.